1092. Shortest Common Supersequence

Hard String Dynamic Programming Leetcode Link

Problem Description

The problem provided requires us to find the shortest string that contains two given strings, `str1` and `str2`, as subsequences. A subsequence of a string `t` is a new string generated from `t` after removing some (can be none) characters without changing the order of the remaining characters. For example, "ace" is a subsequence of "abcde" while "aec" is not. The task is to construct the shortest common supersequence that has both `str1` and `str2` as its subsequences. It is important to note that there can be multiple valid supersequences that satisfy the conditions, and any valid one can be returned.

Intuition

The intuition behind the solution starts by recognizing that this is a classic problem of finding the Longest Common Subsequence (LCS) of the two strings.

1. The LCS assists us in identifying the common base around which we can organize the other characters in the supersequence. Since the LCS is the longest sequence found in both strings, we need to include it only once to cover its occurrence in both `str1` and `str2`.
2. Starting with an empty result string, we move through `str1` and `str2` in parallel, adding characters from both strings to the result. Whenever we find a pair of characters in `str1` and `str2` that are part of the LCS, we only add this character once to the result.
3. To be more practical, we create a dynamic programming table, denoted as `f`, that stores the lengths of LCS for different substrings of `str1` and `str2`. This table can be built by iterating over all characters of `str1` and `str2` where `f[i][j]` represents the length of the LCS between `str1[:i]` and `str2[:j]`.
4. Once we have filled this table, we can backtrack from `f[m][n]` (where `m` is the length of `str1` and `n` is the length of `str2`) to construct the shortest common supersequence by choosing characters from `str1` or `str2` or both when they are the same and they match the LCS character.

By using the dynamic programming approach to find the LCS and careful backtracking to build the actual supersequence, we can construct a supersequence that is the shortest possible combination of `str1` and `str2`, containing both as subsequences.

Solution Approach

The solution utilizes dynamic programming (DP) to compute the shortest common supersequence. Here's how the steps break down in the provided code:

1. A two-dimensional DP array `f` with dimensions `(m+1) x (n+1)` is initialized with zeros, where `m` and `n` are the lengths of `str1` and `str2`, respectively. This array will eventually contain the lengths of the longest common subsequences for all possible substrings of `str1` and `str2`.
2. We fill in the DP table `f` using a nested loop. Here, `f[i][j]` will be filled with the length of the LCS of substrings `str1[:i]` and `str2[:j]`. If `str1[i - 1]` is equal to `str2[j - 1]`, this means the characters match and can be part of the LCS, so we set `f[i][j]` to be `1` plus the length of the LCS at `f[i - 1][j - 1]`. Otherwise, we take the maximum of the lengths of the two possible LCS, by either including the last character of `str1` (`f[i - 1][j]`) or `str2` (`f[i][j - 1]`).
3. After constructing the DP table, we backtrack to build the shortest common supersequence. The list `ans` is used to store the characters of the supersequence in reverse as we'll be starting from the bottom-right corner of the DP table and moving towards the top-left corner.
4. The backtracking logic works as follows:
   - If `i` or `j` reaches `0`, it means we have reached the end of `str1` or `str2`, so we append the remaining characters of the other string.
   - If `f[i][j]` is equal to `f[i-1][j]`, it means the current character of `str1` is not part of the LCS, so we include `str1[i-1]` and move left in the table (decrement `i`).
   - If `f[i][j]` is equal to `f[i][j-1]`, it's the character from `str2` that's not part of the LCS, so we add `str2[j-1]` to the answer and move up (decrement `j`).
   - If none of the above conditions meet, it means both characters from `str1` and `str2` are part of the LCS, so we add either one to the answer and move diagonally up-left in the table (decrement both `i` and `j`).
5. Because `ans` contains the characters in reverse order (from backtracking), we reverse it back to get the correct order of the shortest common supersequence and return it as a string.

This algorithm ensures the generation of the shortest string which is a supersequence of both `str1` and `str2`, using DP for efficiency in finding the LCS, and backtracking to build the supersequence from the LCS.

Example Walkthrough

Let's take two strings `str1 = "abac"` and `str2 = "cab"` to demonstrate how to find the shortest common supersequence using the dynamic programming approach.

1. First, we initialize an empty DP table `f` with dimensions `(4+1) x (3+1)` since `len(str1)` is 4 and `len(str2)` is 3. Each cell in `f` will represent the longest common subsequence (LCS) length for substrings ending at `str1[i-1]` and `str2[j-1]`.
2. We then populate the table `f` as follows:
   - Start with `i = 1` and `j = 1`. If `str1[i - 1] == str2[j - 1]`, set `f[i][j]` to `f[i - 1][j - 1] + 1`. Otherwise, set `f[i][j]` to `max(f[i - 1][j], f[i][j - 1])`.
   - Repeat this process to fill the entire table. Since "abac" has no common characters with "cab" in the first position, the first row and column will be filled with incremental counts.

   The filled DP table `f` would look something like this:

   ```
     c a b
   0 0 0 0
   a 0 1 1 1
   b 0 1 2 2
   a 0 1 2 2
   c 0 1 2 2
   ```

   This table tells us that the length of the LCS for "abac" and "cab" is 2.
3. Starting from `f[4][3]`, we backtrack to construct the supersequence. Initializing `i = 4`, `j = 3`, we go backwards and:
   - Note that the characters `str1[3]` (`c`) and `str2[2]` (`a`) aren't equal. Since `f[4][3] == f[3][3]`, we add "c" from `str1` to our answer and decrement `i` to 3.
   - Now, `str1[2]` (`a`) matches `str2[2]` (`a`), so we add "a" from either string to our answer and decrement both `i` and `j`.
   - At `f[2][1]`, the characters `str1[1]` (`b`) and `str2[0]` (`c`) don't match. Since `f[2][1] == f[1][1]`, we add "b" from `str1` and decrement `i`.
   - Finally, `str1[0]` matches `str2[0]` (`c`), so we add "c" to our answer and decrement each index.

   The result string (`ans`) at this point is "cbac" in reverse order.
4. Reversing `ans`, we get "cabc" which is the shortest common supersequence of `str1` and `str2`.

In this way, by using dynamic programming to determine the LCS of the two strings and carefully backtracking from the end, we have successfully constructed the shortest string containing `str1 = "abac"` and `str2 = "cab"` as subsequences.

Python Solution

```python
class Solution:
    def shortestCommonSupersequence(self, str1: str, str2: str) -> str:
        # Get the lengths of both input strings
        len_str1, len_str2 = len(str1), len(str2)

        # Dynamic programming table f, where f[i][j] will store the length of the
        # longest common subsequence of str1[:i] and str2[:j]
        dp_table = [[0] * (len_str2 + 1) for _ in range(len_str1 + 1)]

        # Build the table in bottom-up manner
        for i in range(1, len_str1 + 1):
            for j in range(1, len_str2 + 1):
                # If characters match, add 1 to the diagonal value
                if str1[i - 1] == str2[j - 1]:
                    dp_table[i][j] = dp_table[i - 1][j - 1] + 1
                # Otherwise, take the maximum value from above or left cell
                else:
                    dp_table[i][j] = max(dp_table[i - 1][j], dp_table[i][j - 1])

        # This will hold the shortest common supersequence characters
        sscs = []

        # Initialize pointers for both strings
        i, j = len_str1, len_str2

        # Trace back from the bottom-right corner of the table
        while i > 0 or j > 0:
            if i == 0:
                # If we've finished str1, add remaining str2 characters
                j -= 1
                sscs.append(str2[j])
            elif j == 0:
                # If we've finished str2, add remaining str1 characters
                i -= 1
                sscs.append(str1[i])
            else:
                # Follow the path of the longest common subsequence
                if dp_table[i][j] == dp_table[i - 1][j]:
                    i -= 1
                    sscs.append(str1[i])
                elif dp_table[i][j] == dp_table[i][j - 1]:
                    j -= 1
                    sscs.append(str2[j])
                else:
                    # If characters match, go diagonally up-left and add the character
                    i -= 1
                    j -= 1
                    sscs.append(str1[i])

        # The sscs list contains the shortest common supersequence in reverse order;
        # reverse it to get the correct sequence
        return ''.join(sscs[::-1])
```

Java Solution

```java
class Solution {
    public String shortestCommonSupersequence(String str1, String str2) {
        int str1Length = str1.length(), str2Length = str2.length();
        int[][] dp = new int[str1Length + 1][str2Length + 1];

        // Calculate the length of longest common subsequence using dynamic programming
        for (int i = 1; i <= str1Length; ++i) {
            for (int j = 1; j <= str2Length; ++j) {
                if (str1.charAt(i - 1) == str2.charAt(j - 1)) {
                    // Characters match, take diagonal value plus one
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // Choose the maximum value from the cell above or the cell to the left
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // Reconstruct the shortest common supersequence from the dp table
        StringBuilder shortestSupersequence = new StringBuilder();
        int i = str1Length, j = str2Length;

        while (i > 0 || j > 0) {
            if (i == 0) {
                // If we have reached the beginning of str1, add remaining characters from str2
                shortestSupersequence.append(str2.charAt(--j));
            } else if (j == 0) {
                // If we have reached the beginning of str2, add remaining characters from str1
                shortestSupersequence.append(str1.charAt(--i));
            } else {
                // Move diagonally if characters match
                if (dp[i][j] == dp[i - 1][j - 1] + 1) {
                    shortestSupersequence.append(str1.charAt(--i));
                    --j;
                } else if (dp[i][j] == dp[i - 1][j]) {
                    // Move up if the character from str1 is part of the supersequence
                    shortestSupersequence.append(str1.charAt(--i));
                } else {
                    // Move left if the character from str2 is part of the supersequence
                    shortestSupersequence.append(str2.charAt(--j));
                }
            }
        }

        // The constructed supersequence is in reverse, so reverse it to get the final string
        return shortestSupersequence.reverse().toString();
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <string>

class Solution {
public:
    // Method calculates the shortest common supersequence of two strings
    std::string shortestCommonSupersequence(std::string str1, std::string str2) {
        int m = str1.size(); // Length of str1
        int n = str2.size(); // Length of str2

        // dp table with dimensions m+1 by n+1, initialized to 0
        std::vector<std::vector<int>> dp(m + 1, std::vector<int>(n + 1, 0));

        // Fill the dp table
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (str1[i - 1] == str2[j - 1]) {
                    // Characters match, increment the length of common subsequence
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // Characters do not match, take the maximum from either the top or left cell
                    dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // Reconstruct the shortest common supersequence from the dp table
        std::string sequence;
        int i = m, j = n;
        while (i > 0 || j > 0) {
            if (i == 0) {
                // If we have reached the beginning of str1, append remaining str2
                sequence += str2[--j];
            } else if (j == 0) {
                // If we have reached the beginning of str2, append remaining str1
                sequence += str1[--i];
            } else {
                // Decide which character to append from either str1 or str2
                if (dp[i][j] == dp[i - 1][j]) {
                    // Coming from top, append str1's character
                    sequence += str1[--i];
                } else if (dp[i][j] == dp[i][j - 1]) {
                    // Coming from left, append str2's character
                    sequence += str2[--j];
                } else {
                    // When characters match, append one of them and move diagonally
                    sequence += str1[--i];
                    --j;
                }
            }
        }

        // Since the construction was from back to front, reverse the string
        std::reverse(sequence.begin(), sequence.end());

        return sequence;
    }
};
```

Typescript Solution

```typescript
function shortestCommonSupersequence(str1: string, str2: string): string {
    const str1Length = str1.length;
    const str2Length = str2.length;
    // Create a 2D matrix to store the lengths of the longest common subsequences
    const dpMatrix = new Array(str1Length + 1).fill(0).map(() => new Array(str2Length + 1).fill(0));

    // Build the matrix with the lengths of the longest common subsequences
    for (let i = 1; i <= str1Length; ++i) {
        for (let j = 1; j <= str2Length; ++j) {
            if (str1[i - 1] === str2[j - 1]) {
                dpMatrix[i][j] = dpMatrix[i - 1][j - 1] + 1;
            } else {
                dpMatrix[i][j] = Math.max(dpMatrix[i - 1][j], dpMatrix[i][j - 1]);
            }
        }
    }

    // Initialize an array to build the shortest common supersequence
    let supersequence: string[] = [];
    let i = str1Length;
    let j = str2Length;

    // Backtrack through the matrix to find the shortest common supersequence
    while (i > 0 || j > 0) {
        if (i === 0) {
            supersequence.push(str2[--j]);
        } else if (j === 0) {
            supersequence.push(str1[--i]);
        } else {
            if (dpMatrix[i][j] === dpMatrix[i - 1][j]) {
                supersequence.push(str1[--i]);
            } else if (dpMatrix[i][j] === dpMatrix[i][j - 1]) {
                supersequence.push(str2[--j]);
            } else {
                supersequence.push(str1[--i]);
                --j;
            }
        }
    }

    // Reverse the array and join it to form the final supersequence string
    return supersequence.reverse().join('');
}
```

Time and Space Complexity

The given Python code defines a method `shortestCommonSupersequence` to find the shortest common supersequence of two strings `str1` and `str2`. To analyze the computational complexities, we need to consider both the time taken to execute the code, which depends on the number of operations performed, and the space required to store the data during execution.

Time Complexity

The time complexity of the algorithm is determined by the nested loop structure which iterates over the lengths of the two input strings `str1` and `str2`. There are two main steps in this algorithm:

1. Filling out the dynamic programming table `f`, which has dimensions $(m + 1) * (n + 1)$, where m is the length of `str1` and n is the length of `str2`. Each cell in the table is filled in constant time, so the time required to fill this table is directly proportional to the number of cells, resulting in a time complexity of $O(m * n)$.
2. Constructing the shortest common supersequence by traversing back from the bottom-right corner of the table to the top-left, appending characters accordingly. The length of this path is $m + n$ in the worst case because each step either moves left or up in the table, and there can be at most m upward moves and n left moves. Therefore, the time complexity for the reconstruction step is $O(m + n)$.

The overall time complexity of the algorithm is the sum of these two parts, predominantly the first since it involves iterating over a two-dimensional array. Thus, the total time complexity is $O(m * n) + O(m + n)$, which simplifies to $O(m * n)$ because $m * n$ dominates $m + n$ for large values of m and n.

Space Complexity

The space complexity is dictated by the space required to store the dynamic programming table `f`, which has $(m + 1) * (n + 1)$ cells, each storing an integer. Hence, the space complexity is $O(m * n)$ to accommodate this table. Additional space is used for the list `ans` which is used to construct the shortest common supersequence. In the worst case, `ans` could grow to a size of $m + n$, when all characters from both `str1` and `str2` are used in the supersequence. However, the space complexity remains $O(m * n)$ because $m * n$ dominates $m + n$.

In summary, the time complexity of the code is $O(m * n)$, and the space complexity of the code is also $O(m * n)$.