Homework 8: Dynamic Programming
Longest Common Substring Problem

November 18, 2024

1 Introduction

Dynamic Programming (DP) is a powerful technique for solving optimization problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. In this report, we explore the transformation of the Dynamic Programming algorithm used for calculating the **Longest Common Subsequence (LCS)** into an algorithm for finding the **Longest Common Substring (LCSstr)** where a substring must be continuous.

2 Data Structures and Recursion

To solve the **Longest Common Substring** problem using dynamic programming, we utilize a two-dimensional table $c[0 \dots m, 0 \dots n]$, where m and n are the lengths of the input strings X and Y, respectively. The entry $c[i][j]$ represents the length of the longest common substring ending at positions i in X and j in Y.

- **Initialization:**
  - Set $c[i][0] = 0$ for all i from 0 to m.
  - Set $c[0][j] = 0$ for all j from 0 to n.
- **Recurrence Relation:**
  - For each i from 1 to m:
    - For each j from 1 to n:
      - If $X[i] == Y[j]$, then:
        $c[i][j] = c[i-1][j-1] + 1$
      - Else:
        $c[i][j] = 0$

Additionally, we maintain two variables:

- maxLength: Tracks the length of the longest common substring found.
- maxI: Records the ending position of this substring in X.

3 Similarity and Differences with Longest Common Subsequence and Lowest Common Substring

Similarity:

- Both problems utilize dynamic programming and a two-dimensional table to store intermediate results.
- Both algorithms involve comparing characters or digits from two inputs and building upon previously computed values.

Differences:

- **Longest Common Subsequence (LCS):**
  - Allows for non-continuous subsequences.
  - When characters do not match, the algorithm chooses the maximum value from the left or top cell.
- **Longest Common Substring (LCSstr):**
  - Requires the substring to be continuous.
  - When characters do not match, the length resets to 0.

4 Algorithm for Longest Common Substring

Below is the algorithm for calculating the length of the Longest Common Substring between two strings X and Y.

Algorithm 1 Longest Common Substring Length Calculation

```
procedure LCSSTR-LENGTH($X, Y$)
    Let $m \leftarrow$ length($X$)
    Let $n \leftarrow$ length($Y$)
    Let $c[0 \dots m, 0 \dots n]$ be a new table initialized to 0
    $maxLength \leftarrow 0$
    $maxI \leftarrow 0$
    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            if $X[i-1] == Y[j-1]$ then
                $c[i][j] \leftarrow c[i-1][j-1] + 1$
                if $c[i][j] > maxLength$ then
                    $maxLength \leftarrow c[i][j]$
                    $maxI \leftarrow i$
                end if
            else
                $c[i][j] \leftarrow 0$
            end if
        end for
    end for
    return $maxLength$, $maxI$, and $c$
end procedure
procedure PRINT-LCSSTR($X$, $maxI$, $maxLength$)
    return $X[maxI - maxLength \dots maxI]$
end procedure
```

5 Time and Space Complexity Analysis

Time Complexity:

The time complexity $T(n)$ of the Longest Common Substring algorithm can be determined by analyzing the number of operations performed:

- **Initialization:**
  - Initializing the table c takes $m \times n$ operations.
- **Main Computation:**
  - Nested loops where the outer loop runs m times and the inner loop runs n times, resulting in $m \times n$ operations.
- **Total Time Complexity:**

$$T(n) = m \times n + m \times n = 2 \times m \times n$$

Space Complexity:

The space complexity is primarily determined by the storage of the table c, which occupies space proportional to $m \times n$.

$$\text{Space Complexity} = m \times n$$

6 Conclusion

The Longest Common Substring (LCSstr) problem extends the principles of Dynamic Programming used in the Longest Common Subsequence (LCS) problem by enforcing continuity in the substring. By modifying the recurrence relation to reset the count when characters do not match, the algorithm efficiently identifies the longest continuous matching segment between two strings. The time complexity of the algorithm is $T(n) = 2 \times m \times n$ and the space complexity is proportional to $m \times n$, where m and n are the lengths of the input strings. The implementation and testing outputs can be found here.