Project Members: Mark Cubi

April 26, 2024

Document Reconstruction Approach

For my project, I first tried to solve the problem with a brute force iterative method. I used two for-loops to traverse through the corrupted file s, and test strings a[j…i] (j being the iterator of the inner for-loop and i being the iterator for the larger for-loop] to see if a was a valid word by passing it to dict, dict(a). If it was, I made sure to add a vector that contained the ongoing sequence of words in the list. If the characters of the words in the sequence matched that of the corrupted, I knew I could fully reconstruct the document.

To speed up the process, I decided to use dynamic programming, particularly tabulation, to solve the problem from a bottom-up approach. I created a dp table [0…s], with s being the size of the corrupted document. The table stored a bool value at each index, which was initialized to false, besides d[0] which was initialized to true. The dp table would store true at dp[i] if a string a[j…i] was a valid word (i.e, it stored that a valid word ends at that position in the document). Another vector would store j’s position (so that the length of the word can be retrieved). I know I could have just made dp store the length of the word at i, but I thought it would take away from its purpose of signifying if a word exists at i or not. The dp table allows the programming to stop the loop early if it finds a word and another word exists before it. For example:

s=[“itwas”]

dp[2]=true (since s[0]+s[1] forms word “it”]

When the loop starts at “s” in “was”, it does not have to progress through “it” because it already knows it is a word. Thus, dp[5]={d(“was”)+dp[2]}

This allows for a general formula dp[i]={d(j…i-1)+dp[j]}

*The algorithm I created did not use a recursive method, but I made sure to include what a recursive method would look like in a decision tree that is in a separate document (the decision tree also follows pretty similarly to my algorithm)*

To determine if the document can be reconstructed, dp[length of document] will determine if it could be fully reconstructed. This is because dp[s] implies there were consecutive words before it in the sequence.

This is also useful when finding the words themselves, as all that is needed is a for-loop that starts at the last index of dp, and then add the sequence of characters and dp[i] based on the length of the substring stored through the vector that stored each words starting position j. The program progresses backwards from word to word till the sequence is complete. This backwards progression also prevents the program from adding correct words that do not lead to a fully reconstructed document.

Ex.

s={“itwas”} does not include “i” because that would lead to sequence “i”,”was” which is not fully reconstructed.

The only issue I had with my program is not being able to account for valid sequences.

Ex.

s={youareaboldone} can be reconstructed to {“you”, “are”, “a”, “bold”, “one”} or {“you”, “area”, “bold”, “one”} are both correct, my program just outputs one of the sequences. My theory is that if dp be a vector<vector<bool>> I could track multiple sequences of correct words.