Homework 9: Dynamic Programming

Data Structures and Algorithms Spring 2020

Sparsh Bansal

1 Question 1

In the minimum edit distance problem, we are given two strings and the goal is to find the minimum number of edits needed to transform $s1$ into $s2$, where a single edit consists of either (a) an insertion of a single character, (b) a deletion of a single character, or (c) a substitution of a single character.

One way to think about dynamic programming equations to solve this problem is to use the bottom-up approach. The sub-problem for calculating each state can be comparison between the two given strings starting from any one end. The result of this comparison can be that the characters are identical, the characters are different, or the strings are of unequal lengths and the character at the particular index does not exist. As we conduct these checks, we can evaluate the 'edit distance,' or how many edits away the strings are from being identical. This particular function will be a case of dynamic programming, since we will update the edit distance for a particular string case, only if it results in being smaller than the previously calculated minimum edit distance.

1.1 Part a

Value function equations can be represented as follows -

- state = index value (checking for unequal input string lengths)
  $V(i, 0) = i$
  $V(0, j) = j$
- state = characters are the same (addition to the edit distance must be 0, since the minimum edit distance should remain the same as calculated in the previous instance of the string up until till the character being compared currently)
  $V(i, j) = V(i - 1, j - 1)$
- state = characters are mismatched (make a decision based on the previously calculated minimum edit distance). The edit functions are unweighted, hence 1 is added at the end to incorporate the cost of calling any of these functions (insertion, deletion or substitution).
  $V(i, j) = 1 + min(V(i - 1, j), V(i, j - 1), V(i - 1, j - 1))$

To argue the correctness of this DP solution using the principal of optimality, we can look at the optimality of the base case and a general case. The base case is optimal, since it corresponds with the index number as per the length of the string. Hence, either there is a match or a mismatch between the characters. For a general case, let S* be the optimal subset of edit distances, such that it stores the minimum number of edits required to reach every possible string $s2$ from string $s1$. Either value i is in S* or not. If it is not, then S* is the optimal subset for items $i + 1..a * b$, such that it stores the minimum edit distance (Here, a equals the length of string $s1$ and b equals the length of string $s2$.). Otherwise, S*-i is the optimal subset for items $i+1..a*b$ such that the minimum edit distance stored in S* is more than or equal to the minimum edit distance represented by i. Therefore, since this DP solution takes the best of both possibilities, it correctly computes the optimal values.

1.2 Part b

The runtime of this function will depend on the length of each of the input strings, since we iterate over all the given possibilities. It is given by $O(a*b)$.

2 Question 2

The goal of the wildcard matching problem is to find if it is possible to use substitution operations into wildcard characters ($*$, in this case, which can represent any possible sub-strings of string $s2$) such that the end result yields $s1$ (which is a fixed string of $a-z$ characters).

To solve this problem using dynamic programming, we can iterate over the string $s1$, comparing it to string $s2$. The sub-problem for calculating the state (Either True or False) can be comparison between the characters. The solution involves indexing into string $s1$ character by character, comparing it to the current index for string $s2$ (starting at 0). The state remains True until each progressive character in string $s1$ is either identical to the the character at the current index for string $s2$, or falls in the $*$ category, where a substitution can be made in order to make the sub strings identical. As soon as this progression is violated, the state is changed to False and the program exits.

2.1 Part a

Value function equations can be represented as follows -

- state = True (for the case when a character in $s1$ corresponds to an identical character in $s2$ or falls under a wildcard character).
  $V(i,j) = True$
- state = False (for all the remaining cases, i.e. when wildcard matching is not possible)
  $V(i,j) = False$

To argue the correctness of this DP solution using the principal of optimality, we can look at the optimality of the base case and a general case. The base case is optimal, since it is when we have iterated through the entirety of strings $s1$ and $s2$, and there has been no violations as described above. Hence, either there is a match or a mismatch between the characters. For a general case, let S* be the optimal subset of Boolean values, such that it stores whether it is possible to use only substitutions in place of wildcard characters to successfully obtain string $s1$ from $s2$. Either value i is True or False. If it is True, then the final decision in S* is the optimal solution for items $i+1..a*b$, such that we can use wildcard substitutions successfully (Here, a equals the length of string $s1$ and b equals the length of string $s2$.). Otherwise, it is False in the optimal subset for items $i+1..a*b$ such that it is not possible to make any number of wildcard substitutions to reach our goal successfully. Therefore, since this DP solution takes the best of both possibilities, it correctly computes the optimal values.

2.2 Part b

As derived in Question 1, Part (b) - the runtime would be within $O(a * b)$, where a is the length of string $s1$, and b is the length of string $s2$. Note: In terms of the algorithm in implementation, we could make the function save time by not going through all possible iterations, since the final decision in this kind of DP relies on whether it evaluated to a False in any one or more iterations. This idea is used in the attached implementation file.

2.3 Part c

Please refer to the separate implementation file.