Homework #9

Megan Ku
Data Structures and Algorithms

April 10, 2020

1. Minimum Edit Distance Problem

Equations and Correctness

Consider two strings m and n. let `x = len(m)` and `y = len(n)`. Let i and j be the current index of strings m and n, respectively. V[i,j] will then be the edit distance between the first i characters of m and the first j characters of n. The base cases assume that one of the strings is empty; that means that the edit distance is simply the length of the non-empty string. We compute the base case for all values of i $\in$ [1,2,...x] and all values of j $\in$ [1,2,...y].

$$V[i, 0] = i$$
$$V[0, j] = j$$

The general value function then considers the cost of editing two strings and returns the minimum.

$$V[i, j] = \begin{cases} V[i-1, j-i] & m[i] = n[j] \\ 1 + \min(V[i, j-i], V[i-i, j], V[i-1, j-i]) & \text{otherwise} \end{cases}$$

Let's now prove the correctness using the principle of optimality. In the base case, we know that the minimum edit distance between a string and an empty string is the length of the longest string, because that would be as many inserts needed to recreate the second string. In the general value function, we know that if two strings are equal, we do not need to edit either string at all. So all sub-strings also have zero edit distance. Otherwise, we need to make one change to account for the misalignment at that index, so we add one to 1 edit count. Either an edit can be made by inserting a character, deleting a character, or replacing one character with another. If a character needs to be added to the first string, then the edit distance is 1 + V[i, j-1] because that step accounts for the edit and then removes it from being considered twice. If a character needs to be removed from the first string, the edit distance is 1 + V[i-1, j], because we account for the cost of the edit and remove that mismatched character from consideration. If the character needs to be replaced, the cost is 1 + V[i-1, j-1] for the same reasons as above. Because our function takes the minimum cost of those three scenarios at each step, we know that the algorithm returns the optimal answer.

b. Runtime

The runtime is O(x*y), where x and y are the lengths of strings m and n, respectively. This is because we are iterating through each character in both strings, so we need to account for both in our runtime.

2. Wildcard Matching Problem

a. Equations and Correctness

Let V[i,j] be whether sub-strings s1[0:i] and s2[0:j] fulfill the wildcard matching pattern. Our base cases then are the following:

$$V[i,0] = \text{False}$$
$$V[0,0] = \text{True}$$
$$V[0,j] = \begin{cases} \text{True} & \text{if the only characters in s2[0:j] are `*'} \\ \text{False} & \text{otherwise} \end{cases}$$

For the general cases, we work bottom-up, comparing each character with the exception that if we run into the '*' character, we check smaller sub-strings.

$$V[i,j] = \begin{cases} V[i,j-1] \text{ or } V[i-1,j] \text{ or } V[i-1,j-1] & \text{if s2[j-1] = `*'} \\ V[i-1,j-1] & \text{if s1[i-1] = s2[j-1]} \\ \text{False} & \text{otherwise} \end{cases}$$

To argue correctness, let's first look at the base case. We know that for any non-empty s1, if s2 is empty, we will return False. We also know that an empty string and another empty string will return True. We then know that if s1 is empty and s2 is only comprised of '*', then V[0,j] will be True. Now, for every s1 and s2 sub-string that we match up, we can check the last character in both strings. If they are equal, then the answer to whether the whole string is equal is if V[i-1, j-1] is true. If the character in s2 is a '*', then we should check the following cases: the '*' represents only one character (V[i-1,j-1]), it represents multiple characters (V[i-1,j]), or it represents an empty string (V[i, j-1]). Since only one of those scenarios has to return true, which is why the union of them will return the correct output. If neither of these scenarios occur, then s2 cannot be made into s1, which is why we return False. Because the algorithm considers all of these cases, we know the algorithm is correct.

b. Runtime

The runtime of this DP algorithm takes O(len(s1)*len(s2)). This is because we iterate through the characters of both s1 and s2 to check if the wildcard pattern holds.

c. Implementation

See mku.py.