Homework 9

Sam Daitzman // DSA Spring 2020

Exercise 1: Minimum Edit Distance Problem

> In this question, you will develop (but not implement) a dynamic programming approach to solve the minimum edit distance problem. In the minimum edit distance problem, you are given two strings s1 and s2 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.
>
> For example, if s1 = 'cake' and s2 = 'cat' then the minimum edit distance would be two edits: deleting 'e' and substituting 't' for 'k' (or equivalently deleting 'k' and substituting 't' for 'e').

Exercise 1A: Dynamic Programming Equations & Principle of Optimality

> Give a set of dynamic programming equations to find the minimum edit distance between two strings. Be sure to state what your value function calculates. Then, in 3-5 sentences, argue correctness of your DP solution using the principle of optimality.

We're trying to find the minimum possible edit distance between two strings s1 and s2. I'll call the length of string s1 m, and the length of s2 n. To convert this to a dynamic programming problem, I'll break the overall problem down into subproblems. I'll begin at the last letter and work backwards, calling my dynamic programming function recursively. In my base case, either s1 or s2 may be empty. In this case, if s1 is empty, we will use n (as in, the length of s2 at the current recursion layer, not the original s2 string) inserts to insert each character from the second string. And if s2 is empty, we will use m (as in, the length of s1 at the current recursion layer, not the original s1 string) removes to remove every character from s1 so the strings match.

My dynamic programming equations need to account for all possible situations so we can satisfy the principle of optimality. I'll choose a value function designed to compute the current edit distance. I'll call that function $D_i(m, n)$. It is a piecewise function that will take the following values:

1. In our base case where the first string is empty: $D_i(m, n) = n$ and we can simply return that.

2. In our base case where the second string is empty: $D_i(m, n) = m$ and we can simply return that.
3. In the case where the last two characters are the same: $D_i(m, n) = D_i(m-1, n-1)$, meaning we recurse down a layer with the final two letters excluded and do not increase the cost (because no operations were needed on these letters).
4. If the last two characters of the strings at this level of recursion are not equal, we will need to try each of our three valid operations and take the minimum possible value. So our cost function will be:

$$D_i(m, n) = 1 + min(D_i(m-1, n), D_i(m, n-1), D_i(m-1, n-1))$$

This is correct because it follows the principle of optimality. I'll consider the base case first, where we will find that one string or the other is empty, or that our two characters match. For each of these cases:

1. If one string is empty, the minimum number of operations to fill it or clear the other will be the size of the other, so we return that. This case is fulfilled because in all versions of it, we'll return an optimal solution.
2. If our two characters match, we have nothing to do, so we'll return 0 and correctly continue.

In every further iteration, we'll continue with one of several possibilities:

1. In the case where either string is empty, we continue the same way as in the base case, so this case is fulfilled.
2. In the case where the last two characters are the same: we continue along in the word and do not add any cost, which will allow us to continue towards the most optimal solution, so this case is fulfilled.
3. If the last two characters of the strings at this level of recursion are not equal, we will need to try each of our three valid operations and take the minimum possible value. We'll recurse, add 1 to the cost and return the minimum of the three possible subproblems. Since we try all three allowable operations and choose the minimum possible cost, and each operation adds a cost of 1, we correctly return the minimum possible value and this case is fulfilled.

In the end, we'll return the sum passed up through this tree, and it will preserve the optimal case across each possibility and test every possibility. Thus, all cases are fulfilled according to the principle of optimality.

Exercise 1B: Runtime

What is the runtime of calculating your DP solution?

Since we end up computing a maximum of 3 constant-time operations for each value, across a maximum of $m * n$ values, our runtime is $O(m * n)$.

Exercise 2: Wildcard Matching Problem

In this question, you will see and implement an example of dynamic programming for a problem that does not involve optimization. In the wildcard matching problem, you are given a pattern string s1 and a wildcard string s2. While s1 is a fixed string of a...z characters, s2 may contain one or more wildcard characters * which represent any possible substring (including the empty string). The goal of this problem is to find whether there exists a substitution into the wildcard characters such that the end result yields s1.

For example, if s1 = 'lemondrop' and s2 = 'l*dr*p' *the answer would be True since we can substitute in 'emon', 'o', and '' for each of the wildcard characters, respectively. However, if s2 = 'lem*m*dr*p*'* the answer would be False because there is no possible wildcard match that would generate s1.

Exercise 2A: Dynamic Programming Equations

Give a set of dynamic programming equations to find whether or not there is a wildcard match between two strings s1 and s2. Be sure to state what your value function represents. Then, in 3-5 sentences, argue correctness of your DP solution using the principle of optimality. **Hint**: Your value function should evaluate to True or False.

S1 IS NORMAL TEXT // S2 IS THE WILDCARD PATTERN STRING

We want to compare some valid string of letters s1 to a wildcards string s2, which can contain some number of * characters that can be replaced by none or many characters. We'll convert this to a dynamic programming problem by choosing a value function that can have the value true or false. Again, I'll solve this problem with a recursive dynamic programming call. We'll define a value function $W_{i,j}(m, n)$ with m being the length of string s1 and n being the length of string s2. For each character in the wildcard string s2:

1. If the character is `"*"`, we have two cases, and if either is true we take the value to be true and continue on. The value here is $OR(W_{i,j}(m-1, n), W_{i,j}(m, n-1))$ or, in words, `true` only if the character either is a wildcard or identical to the current index in the other string:
   a. We can continue on past the character, basically ignoring it.
   b. We can use the character to match one or more characters, therefore continuing on.
2. If the characters in both string at our current location match, we continue and look at other values, so our value function will be $W_{i,j}(m, n) = W_{i,j}(m-1, n-1)$

3. If these situations are not satisfied, our value function is `false`.

Throughout my implementation in all subproblems, wherever we see a potential match, we know that it must either correspond to a matching value or a wildcard match. In the case that it corresponds to a wildcard match, we know that we'll continue onwards as expected because we'll continue iterating having only found truthy values. In the case where the strings do not match correctly and cannot be substituted by a wildcard, we'll simply set our value function to false, and our eventual return will be false (in implementation, I actually return False immediately). Thus, all cases are satisfied.

Exercise 2B: Runtime

> What is the runtime of calculating your DP solution?

As before, we calculate a maximum of $m*n$ cells, and computing each cell consists of several constant-time operations when we include the fact that cells are cached across subproblem function calls/recursions so the runtime is constrained to $O(m * n)$.

Exercise 2C: Implementation and Testing

> Implement and test your dynamic program. For your test function, construct 5-7 test cases that you think contain different possible structures.

See `./sdaitzman.py`.

Questions/Notes

- I noticed in many reference algorithms similar to the dynamic programming solutions I implemented, the functions include some subscripts (e.g. $W_i(m, n)$). Does this refer to the row/column? Does it refer to the iteration? Is it just a way of indicating that it varies across values? I wrote the equations the way that made sense to me but I'm wondering if there's a convention here to follow.
- Note to self: I feel very fuzzy about implementing dynamic programming algorithms and should go back over those lecture notes. I leaned pretty heavily on similar reference implementations for this homework.