CSCI3104
Spring 2023

Recitation 10: Dynamic programming

Question 0. See Michael Levet lecture notes pp. 105 - 110.

Question 1. You are given a string s with n elements and asked to convert it into a palindrome. You are allowed to insert characters at various positions of this string.

For example, to make a palindrome out of the string "BLAIR", we proceed as follows: 1) insert characters "RIA" at position 0, 2) insert character "B" at position 2. The result is the string "RIABLBAIR" which is a palindrome.

There are other ways to make "STAIR" (what is a trivial one?) a palindrome but we are interested in finding minimum number of insertions we need to implement to turn a given string into a palindrome.

Formulate a dynamic programming algorithm for finding the smallest number of such insertions.

(i) Write a recursive definition of the function MinPalindromeIns(s) which counts the minimum number of insertions needed to make a string s a palindrome.

(ii) How do you use the recurrence above to construct a memo table? How do you fill in this table? What is the runtime complexity? How do you check the solution?

Answer: Compare the first and last symbols, i and j, of the current (sub)string. If $i = j$, increase i by 1 and decrease j by 1, i.e., look at the substring ranging from the position $i+1$ to $j-1$ at the next iteration. If $i \neq j$, insert a symbol at the position specified by j (i) such that the inserted character will be equivalent to i (j), and count this insertion with 1 cost. After this, look at two different substrings for the next iteration: one with indices $[i+1, j]$, and the other with indices $[i, j-1]$. MinPalindromeIns(s) repeats these steps until the base case is reached, this happens for string with length $0, 1$ (see Part B for implementation with a memotable).

So the recursive process to count the minimum number of inversions $c[i, j]$ needed to form a palindrome from a given string is characterized by:

$$c[i,j] = \begin{cases} c[i+1, j-1] & \text{if } S[i] = S[j] \\ \min(c[i+1, j], c[i, j-1]) + 1 & \text{otherwise} \end{cases}$$

For memoization, for a given string s with length n, we proceed with constructing a table of size $n \times n$, whose i-jth entries consist of $c[i, j]$. From the above, we know that $c[i, j] = 0$ if $i \geq j$ $\forall i, j > 0$, so we fill this table diagonally, knowing that the lower triangular part of the table will consist of zeroes. The time complexity of this algorithm is $O(n^2)$.

```
def min_insert(s):
    n = len(s)
    memo = np.zeros([n, n])
    for y in range(1, n):
    x = 0
        for t in range(y, n):
            if s[x] == s[t]:
                memo[x][t] = memo[x+1][t+1]
            else:
                memo[x][t] = min(memo[x][t-1], memo[x+1][t]) + 1
            x +=1
    return memo[0][n - 1]
```

Calling the function above results in looking at the entry characterized by memo[0][n - 1], thus providing the solution to the problem.

Question 2. The edit distance $d(x, y)$ of two strings of text, $x[1 \ldots m]$ and $y[1 \ldots n]$, is defined as the minimum possible cost of a sequence of transformation operations which convert a given string $x[1 \ldots m]$ into another string $y[1 \ldots n]$. Show that the problem of calculating the edit distance $d(x, y)$ exhibits optimal substructure.

Answer: We must show that computing edit distance for strings x and y can be done by finding the edit distance of subproblems.

Define a cost function

$$c_{xy}(i, j) = d(x, y[1 \ldots i] \| x[j + 1 \ldots m])$$

where $c_{xy}(i, j)$ is the minimum cost of converting the first j characters of x into the first i characters of y. Then $d(x, y) = c_{xy}(n, m)$. Consider a sequence of operations $S = \langle o_1, o_2, \ldots, o_k \rangle$ that transforms x to y with cost $C(S) = d(x, y)$. Let S_i be the subsequence of S containing the first i operations of S. Let z_i be the auxilliary string after performing operations S_i, where $z_0 = x$ and $z_k = y$.

Using this construction, we want to prove that if $C(S_i) = d(x, z_i)$, then $C(S_{i-1}) = d(x, z_{i-1})$. We prove this by contradiction using cut and paste.

Assume that $C(S_{i-1}) \neq d(x, z_{i-1})$. There are two cases, $C(S_{i-1}) < d(x, z_{i-1})$ or $C(S_{i-1}) > d(x, zi - 1)$. If $C < d(x, z_{i-1})$, then we can transform x to z_{i-1} using operations S_{i-1} with lower cost than $d(x, z_{i-1})$, which is a contradiction. If $C(S_{i-1}) > d(x, z_{i-1})$, then we could replace S_{i-1} with the sequence of operations S' that transforms x to z_{i-1} with cost $d(x, z_{i-1})$. Then the sequence of operations $S' \cup o_i$ transforms x to y with cost $C(S' \cup o_i)$

$$\begin{aligned} C\left(S' \cup o_i\right) &= d(x, z_{i-1}) + C(o_i) \\ &< C(S_{i-1}) + C(o_i) \\ &= C(S_i) \\ &= d(x, z_i) \end{aligned}$$

This means that $d(x, z_i)$ is not the edit distance between x and z_i, which is yet another contradiction. Therefore our assumption that $C(S_{i-1}) \neq d(x, z_{i-1})$ must be wrong, and the edit distance exhibits optimal substructure.