Homework 1

CS 4390/5390
Fall 2019

Due: 18 September 2019

This homework is worth 5 points out of the total 25 points of homework in the class.

1. **(2 points)** *Give an algorithm that takes in two string α and β, of length n and m, and finds the longest suffix of α that exactly matches a prefix of β. The algorithm should run in $O(n+m)$ time.*

   algorithm

   (a) construct a new string $S = \beta\$\alpha$, where \$ is a character that is not in either α or β.

   (b) compute the Z-values (longest prefix-length) for each index in the string S.

   (c) make a linear scan for i from $m+2$ to $m+n+1$, return the first index i such that $Z_i + i = m + n + 1$.

   The Z_i value for each index contains the longest prefix of the string that matches the suffix of the string starting at i (that is $S[1...Z_i] = S[i...(i+Z_i)]$). Because of the way our particular S is constructed, we know that for any $i > m$, $Z_i < m$ since it cannot match the inserted \$. This means the value is the exactly the length of substring from α that matches a prefix of β. Then we need to check that that substring is a prefix, and this is done by ensuring that the substring from α contains the last position of S. The condition in Step 1c ensures that we are matching the suffix of α with a prefix of β. The order in which the scan is done ensures that we return the longest such prefix/suffix pair since we are comparing in decreasing order of suffix size.

   The to construct S in Step 1a takes $O(m+n)$-time. We know that calculating all of the Z-values in Step 1b (even-though they are not all used) takes $O(|S|) = O(m+n)$-time. Finally the linear scan in Step 1c can take worst-case $O(n)$-time. This means the total running time is $O(m+n)$ time which satisfies the question.

2. **(2 points)** *Given two strings of length n characters each and an additional parameter k. In each string there are $n-k+1$ substrings of length k (we will later call them*

k-mers), and so there are $\Theta(n^2)$ pairs of substring, where one substring is from one string and one is from the other. For a pair of substrings, we define the match-count as the number of opposing characters that match when the two substrings of length k are aligned. The problems is to compute the match-count for each of the $\Theta(n^2)$ pairs of substrings from the two strings. Clearly the problem can be solved with $O(kn^2)$ operations (character comparison plus arithmetic operation). But by better organizing the computations, the time can be reduced to $O(n^2)$ operations. (Note, this problem can be solved without the Z algorithm or other fancy tools, only thought.)

algorithm

(a) create a new array A of size $(n-k+1) \times (n+k)$ which will contain the final match-count values.

(b) for $jS : 1 \to n-k+1$ (this will iterate the shift of the second string)

i. calculate $A[1, jS]$ manually by comparing each pair of bases in $s_1[1...k]$ with those in $s_2[jS...(jS+k)]$.

ii. for $\ell : 2 \to n+k-1$ (this iterates the offsets for the k-mers) calculate

$$A[\ell, j] = A[\ell-1, jM] - (s_1[\ell-1] == s_2[jM]) + (s_1[\ell+k] == s_2[jP]),$$

where $j = (jS+\ell)\%n, jM = (jS+\ell-1)\%n, jP = (jS+\ell+k)\%n$, and the $==$ operator is 1 if the characters match, and 0 otherwise.

(c) return $A[1...(n-k+1)][1...(n-k+1)]$.

We can prove that the correct answer is returned by induction. **Base:** For each $1 \le p \le n-k+1$, we know that we calculate the match-count for the k-mers $s_1[1...k]$ and $s_2[p...(p+k)]$ correctly since we calculate it naïvely in Step 2(b)i. **Inductive step:** Sssume we know the correct match count for the k-mers starting at i and j (written as the pair (i, j)) $s_1[i...(i+k)]$ and $s_2[j...(j+k)]$, call it MC. We also know that the k-mers starting at $(i+1, j+1)$ share $k-1$ bases from the pair starting at (i, j), the only differences are the characters matched $s_1[i]$ and $s_2[j]$ then $s_1[i+k+1]$ and $s_2[j+k+1]$. This means the match count for $(i+1, j+1)$ will only differ by the counts contributed by those 4 characters, and thus calculate it by the equation in Step 2(b)ii. At some points in the algorithm we calculate the match for k-mers starting in s_2 starting at $i > n-k+1$, in these cases, our algorithm calculates a wrapped k-mer (i.e. $s_2[i...n] \cdot s_2[1...(k-i)]$) but those values are not returned and by the previous argument will still calculate the correct value for $i = (n+1)\%n = 1$.

Creating the array in Step 2a takes $O(n^2)$-time. The loop in Step 2b runs $n-k+1$ times, and in each iteration the running times are $O(k)$ for Step 2(b)i and then a loop that runs an $O(1)$ operation $O(n)$ times. This means Step 2b has a total running time of $O(n^2)$. Finally Step 2c runs $O(n^2)$-time. This means the total running time is $O(n^2)$.

3. **(1 point)** *Given a set $\mathcal{S}$ of k strings, we want to find evey string in $\mathcal{S}$ that is a substring of some other strings in $\mathcal{S}$. Assuming the total length of all of the strings is n, give an $O(n)$-time algorithm to solve this problem.* **algorithm**

   (a) create a generalized suffix tree $\mathcal{T}$ that contains all of the sequences in $\mathcal{S}$.

   (b) for each sequence $S \in \mathcal{S}$

      i. find the leaf ℓ and its parent p_ℓ that maps to the full string S in $\mathcal{T}$

      ii. if the edge label for (p_ℓ, ℓ) is only the string "\$" or if ℓ has more than one label return S.

   For each string S, if its a full substring of some other string it will have one of two conditions when you search for all of S in $\mathcal{T}$: (1) it will match the suffix of some other string, in which case the leaf will be both the leaf of the full string S, and the suffix from the other string, in which case, the leaf will have two labels; or (ii) some other string will contain it fully so the prefix of a suffix will match, in which case the parent of the leaf will match that suffix-prefix, then the tail character will be on a single character edge label. If S is not a substring the leaf label will be of length greater than 1 and it will have only one label.

   Step 3a will take $O(n)$-time to create since the total string length is n. The time required for Step 3(b)i is $O(|S|)$, and amortized over all the iterations of Step 3b is $O(n)$. Finally, the checks in Step 3(b)ii will take $O(1)$-time each, and since there are $|\mathcal{S}| < n$ sequences, the total time for that step is $O(n)$. Therefore the whole algorithm runs in $O(n)$-time.