15-351 / 15-650 / 02-613 Homework #5: Suffix trees/arrays, DP, and network flow

Due: Sunday, Nov. 1 by 11:59pm

You may discuss these problems with your current classmates, but you must write up your solutions independently, without using common notes or worksheets. You must indicate at the top of your homework who you worked with. Your write up should be clear and concise. You are trying to convince a skeptical reader that your answers are correct. Avoid pseudocode if possible. Your homework should be submitted via **GradeScope** as a typeset PDF. A LaTeX tutorial and template are available on the class website if you choose to use that system to typeset. For problems asking for an algorithm: describe the algorithm, an argument why it is correct, and an estimation of how fast it will run. Use O notation for running times.

1. A <u>k-mismatch palindrome</u> is a string xy where, $|x| = |y|$ and reverse(y) and x are the same in all but at most k positions. Give an $O(kn)$-time algorithm to find all the maximal k-mismatch palindromes in a string S of length n.

   Solution: We will first iterate over where the center of the k-mismatch palindrome is. Denote this location i (it means x ends at location $i-1$, and y starts at location i). For each center location i, we will determine the longest palindrome.

   We will assume we have constant time access to function LCP(i, j), as the longest common prefix of $S[i \dots]$ and $S^R[n-1-j \dots]$. The idea is similar to the `checkMismatch` function described in lecture note 10. We start with $l = \text{LCP}(i, i-1)$ as the current half length of the palindrome, and $c = 0$ as the current number of mismatches. As long as $c < k$ and $l < \min(i, n-i)$, we do the following: increment c by one, and l by $1 + \text{LCP}(i+l+1, i-l-2)$. Intuitively, current palindrome starts at $i-l$ and ends at $i+l-1$ (both ends inclusive), so the next mismatch to jump over is at location $i-l-1$ and $i+l$ respectively. This algorithm runs in time $O(k)$ for each center location i, so total time complexity is $O(nk)$.

2. Let x be a string of length n. There are $O(n^2)$ substrings of x. Show how to count the number of <u>distinct</u> substrings of x in $O(n)$ time.

   Solution: The one-line solution is that the number of substrings is the sum of number of characters over every edge of the suffix tree, excluding \$.

   To see why this is true, note that each substring is a prefix of a suffix. We can assume a sequential process of adding suffixes of S to a trie (which would eventually become the suffix tree of S), and after adding each suffix, we want to know how many of its prefixes are not prefixes of existing suffixes (in the trie) already. This is exactly the number of new nodes (excluding \$ nodes) in the trie from adding this suffix, so in the end total number of distinct substrings would equal the size of the suffix trie.

3. Suppose you are given a string s of length n. Describe an $O(n)$-time algorithm to find the longest string t that occurs both forwards and backwards in s. Your algorithm must use suffix trees or generalized suffix trees.

   For example: If $s =$ `yabcxqcbaz`, your algorithm should return $t =$ `abc` or $t =$ `cba` because both `abc` and its reverse `cba` occur in s and no longer such string exists.

   Solution: We make generalized suffix tree of s and reverse of s. Then, we search for the internal node with maximum depth that has end symbols from both strings, and the string represented from the root to that node is non-overlapping (this can also be checked in $O(1)$ time for every node).

4. You are given a rooted tree T of n nodes, where every node i is associated with a weight w_i. Note that w_i can be negative. Your task is to select a subset of nodes to maximize the their total weight. However, if you select node u, then you can't select any of u's descendants. Design an $O(n)$-time dynamic programming algorithm to find the maximum total weight.

Solution: Let $f(i)$ be the maximum weight we can get from the subtree rooted at i. The boundary case is that, for any leaf i, we choose if we pick it:

$$f(i) = \max\{0, w_i\}.$$

The recurrence is

$$f(i) = \max\left\{w_i, \sum_{j \in \text{child(i)}} f(j)\right\},$$

where we choose whether to pick this node. And the final answer is $f(\text{root})$.

In addition to the weight, we store at each state the choice that maximizes the weight, i.e., the results of argmax. It also works if we use DFS after we solve all subproblems to find the optimal subset.

The correctness can be proved by induction, just as the recurrence. The runtime is $O(n)$ because every node appears $O(1)$ times on the right hand side of all recurrences and each time it takes $O(1)$ time.

5. A k-cover $C_k(S, T)$ of a string T is a set of substrings of the string S, each of length $\geq k$, such that T can be written as the concatenation of substrings in $C_k(S, T)$. Give a $O(|S| + |T|)$-time dynamic programming algorithm to compute a k-cover $C_k(S, T)$ given strings S and T and integer k (or report "none" if no such k-cover exists).

Solution: There are two parts to this problem, and the solution below is not unique: There are other ways to do it.

Assignment Project Exam Help

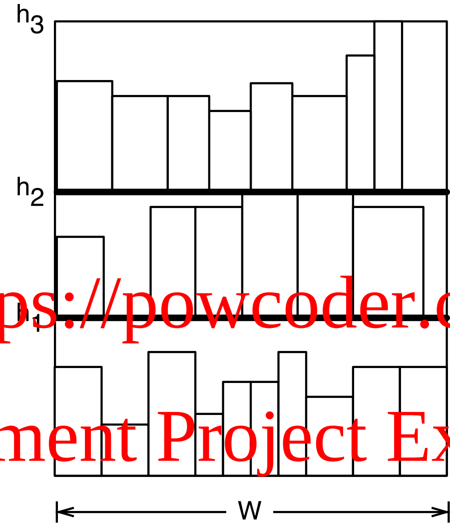
First, for each i, we want to determine the largest j such that $T[i \ldots i + j]$ is a substring of S, denoted $A[i]$. We build a generalized/colored suffix tree of S (colored green) and T (colored red). We first calculate a value $D[x]$ for each node of the suffix tree, which is defined as the longest prefix of the string from root to node x, that is a substring of S. This can be calculated by a depth-first search on the suffix tree. The root has $D[x] = 0$. Let p be the parent of x on the tree, we have $D[x] = D[p]$ if the edge connecting p to x is a pure red edge, and $D[x] = D[p] + w$ otherwise (w is the number of characters on the edge). We now calculate $A[i]$: locate the end node for each $T[i \ldots]$ and set $A[i]$ to the D value on this node.

https://powcoder.com

The second part is a dynamic programming instance (note that a greedy algorithm will not work). Let $F[i]$ denote whether there is a k-cover for $T[i \ldots]$ (we do this in the less intuitive order because of how $A[i]$ is defined). We have $F[n] = 1$ as boundary condition, where n is length of T. If a k-cover exists for $T[i \ldots]$, one of the substrings of S must match $T[i \ldots i + j]$ for some $j \geq k$. As we derived before $j \leq A[i]$, so we have $F[i] = 0$ if $A[i] < k$ and $F[i] = \max_{k \leq j \leq A[i]} F[i + j]$ otherwise. This can be calculated in linear time by maintaining a suffix sum: $Q[n] = 0$, $Q[i] = Q[i + 1] + F[i]$ and $F[i] = \mathbf{1}(Q[i + k] - Q[i + A[i] + 1] \geq 1)$ if $A[i] \geq k$. To report a solution (usually called a backtracking process), we will need to know the argmax in the expression of $F[i]$, which can be done by maintaining another variable $P[i]$ denoting the minimal j such that $j \geq i$ and $F[j] = 1$.

Add WeChat powcoder

6. You are given a list of books $b_1, \ldots, b_n$ in alphabetical order. The height of each book is given by $h(b_i)$ and the width is $w(b_i)$. You are designing a bookcase of width W to store these books, in alphabetical order, and you want the bookcase to be as short as possible. Design a dynamic programming algorithm to compute the height of the shortest bookcase that will hold these books. An example non-optimal solution of height $h_1 + h_2 + h_3$ is given below:



Solution: For each i, denote by A_i the least book index $j \in \{1, \dots, i\}$ such that the total width of books $b_j, \dots, b_i$ doesn't exceed W. This can be solved for each i by enumerating j from i down to 1. Let $f(i)$ be the minimum height of the bookcase that holds books $b_1, \dots, b_i$. The boundary case and the recurrence are

$$f(0) = 0, \quad f(i) = \max_{j:A_i \le j \le i} \{f(j-1) + \max_{k \in \{j,\dots,i\}} h(b_k)\}$$

In words, if we put books $b_j, \dots, b_i$ to one level and put other books into other levels, the height of this level is the second term (the max) and the total height of other levels is $f(j-1)$. We solve the max term from $j = i$ down to A_i and use a running max to accelerate the max term: $\max_{k \in \{j,\dots,i\}} h(b_k) = \max\{h(b_j), \max_{k \in \{j+1,\dots,i\}} h(b_k)\}$, so every j takes $O(1)$ time. There are n subproblems and each takes $O(n)$ time to solve, so the overall runtime is $O(n^2)$.

Before we show the acceleration, we define $g(j,i) := \max_{k \in \{j,\dots,i\}} h(b_k)$ and make several observations. Observation 1: A_i is monotonically non-decreasing in i, which means we can solve A_i for all i in $O(n)$ time. Observation 2: $g(j,i)$ is also non-decreasing in i, and non-increasing in j. Observation 3: The immediate result of observations 1 and 2, and the fact that $f(i-1) + g(i,i) \ge f(i-1)$ is that $f(i)$ is non-decreasing in i. Observation 4: Because $g(j,i)$ is a piecewise constant function and is non-increasing in j, and because $f(i)$ is non-decreasing in i, we consider j only if either $j = A_i$ or $g(j-1,i) > g(j,i)$.

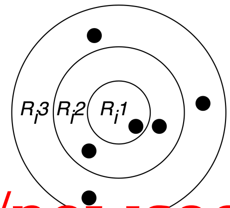
So, we use a queue to maintain the candidate indices in observation 4, and put their values $(f(j-1)+g(j,i))$ in a heap. Specifically, the queue is initially empty, and elements in it are always increasing indices with decreasing heights from left to right. So the value of $g(j,i)$ is simply the height of the next book in the queue. When we move to $f(i)$ from $f(i-1)$, we remove elements from the left that are less than A_i, remove elements from the right whose height is less than or equal to $h(b_i)$, and then append i to the right. As we remove/add elements from/to the queue, we also remove/add corresponding elements in the heap. To solve $f(i)$, we simply query the heap for the minimum value. Because every book is added and removed exactly once, the overall runtime is $O(n \log n)$.

This problem is included in LeetCode (ID 1105).

7. (Network Flow) You are deploying n cheap temperature-measurement devices in the field, with device t_i at coordinates (x_i, y_i), measured in meters from some arbitrary point. These devices record their temperature over several weeks. The devices are likely to fail so you want to design a system to back up the data they have collected in the following way: Each device has a radio transmitter that can reach d meters. When a device t_i senses it is about to fail, it will transmit its data, and that data should reach at least k other devices. Each device can serve as the backup for at most b other devices.

   (a) Design a polynomial-time algorithm to determine whether the given positions of the devices meets the requirements and, if it does, to output the set B_i of k back up devices for every device.

(b) Suppose for every device t_i, we are now given a collection of sets R_i^d for $d = 1, \ldots, k$, where R_i^d contains the set of devices that are at distance ring d from t_i. See figure below:



We add the following requirement: each of the devices in the backup set B_i for device t_i must come from a different ring R_i^d. (That is, we need a very close device from ring R_i^1 and a slightly farther device from R_i^2, etc.) Give a polynomial time algorithm to find the backup sets that meet this requirement.

Solution:

(a) We build a graph $G = (V = \{s, t\} \cup T \cup T', E)$ where $T := \{t_i : i \in [n]\}$ consists of devices as sensors and $T' := \{t'_i : i \in [n]\}$ is the set of devices as backups. The set of edges is

$$
\begin{aligned}
E &:= E_{s \to T} \cup E_{T \to T'} \cup E_{T' \to t} \\
E_{s \to T} &:= \{(s, t_i) : \forall t_i \in T\} \\
E_{T \to T'} &:= \{(t_i, t'_j) : \text{device } t_i \text{ can transmit its data to device } t_j\} \\
E_{T' \to t} &:= \{(t'_i, t) : \forall t'_i \in T'\}
\end{aligned}
$$

Each edge in $E_{s \to T}$ has capacity k because every device needs at least k backups. Each edge in $E_{T \to T'}$ has capacity 1 because we can't store more than one copy of one device in the same backup device. Each edge in $E_{T' \to t}$ has capacity b, because each device can serve as the backup for at most b other devices.

And then we run the Ford-Fulkerson algorithm. If the value of the max flow is less than nk, then we're sure these positions doesn't meet the requirements. Otherwise, we set B_i be the set of devices in T' that receive flow from $t_i \in T$.

Correctness Firstly, the output backup allocation is always feasible. Conversely, assume there is a feasible backup plan. We remove backups in the plan until every device has exactly k backups, and now the plan is also feasible. We can easily construct a flow with value nk, and therefore our algorithm will find these backup sets.

The number of edges is $O(n^2)$ and the capacities of edges are integers. So the runtime is $O(n^3 k)$.

(b) We build a graph $G = (V = \{s, t\} \cup T \cup T', E)$. This time, T' remains the same, while T is $\{t_i^d : i \in [n] \wedge d \in [k]\}$. The edge set is

$$
\begin{aligned}
E &:= E_{s \to T} \cup E_{T \to T'} \cup E_{T' \to t} \\
E_{s \to T} &:= \{(s, t_i^d) : \forall t_i^d \in T\} \\
E_{T \to T'} &:= \{(t_i^d, t'_j) : \text{device } t_i^d \text{ can transmit its data to device } t_j\} \\
E_{T' \to t} &:= \{(t'_i, t) : \forall t'_i \in T'\}
\end{aligned}
$$

The weight of any edge in $E_{s \to T}$ is one, and any other weights are unchanged. The proof of correctness is very similar. Now in this graph, the number of edges is $O(n^2 k)$, and the overall runtime is $O(n^3 k^2)$.