Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2024

Tries—Tuesday, November 25 / Wednesday, November 26

Readings

- Lecture Notes Chapter 24: Tries

Review

A trie is a tree-based data structure that stores strings to support information re**trie**val. Tries are primarily useful when we need to repeatedly query a fixed text because it allows us to pre-process this text such that each subsequent query is fast, offsetting this initial cost of building the trie.

Standard Trie: In a standard trie, each root to leaf path corresponds to some string inserted into the trie. If the total length of all strings inserted into the trie is n, then a standard trie takes $O(n)$ time to build (using an incremental algorithm) and uses $O(n)$ space as well.

Patricia/Compressed Trie: A compressed trie is a trie where we guarantee that every internal node has at least two children by compressing branches/chains of single-child nodes into a supernode. If the total length of all strings is n and we have s strings, then a compressed trie takes $O(n)$ time to build but only uses $O(s)$ space, since the tree is now at least as full as a full binary tree (which has $O(s)$ nodes if it has s leaves).

Suffix Trie: A suffix trie is a trie where the strings are all the suffixes of a string S. Using an incremental algorithm, we can build a suffix trie in $O(|S|^2)$ time, but we can actually also do it in $O(|S|)$ time using Ukkonen's Algorithm; however, the details behind how this works are outside the scope of CIS 121. A compressed suffix trie uses $O(|S|)$ space.

Problems

Problem 1

Given a set of N strings, design an efficient algorithm to find the longest common prefix between any two strings. What is the running time of your algorithm?

Solution

Algorithm: Initialize a variable `count` to 0 to keep track of the length of the longest prefix, and initialize a variable `prefix` to Null to keep track of the actual prefix to return. Add all N strings to the trie as follows: For each string inserted, track the maximum depth into the trie before we have to initialize any new nodes. If this depth is greater than `count`, then update it to be this new depth and then update `prefix` to store the substring corresponding to this new maximum depth. After we have added all N strings as described, return `prefix`.

Proof of Correctness: Note that for two strings w_1 and w_2 to share a common prefix, the insertion of the second string, WLOG w_2, must involve "traversing" a path from the root of the trie to some node v, such that the terminal node of w_1 is a descendant of v. The correctness of our algorithm follows directly from this: if we track the maximum depth d into the trie before we have to initialize any new nodes while inserting strings, then this depth d is the length of some common prefix. Therefore, updating `prefix` if

we find a greater depth ensures that we return the longest common prefix between any two strings at the end.

Runtime Analysis: The runtime of this algorithm is equivalent to the construction time of a trie, since the "modifications" we make — tracking the maximum depth and updating variables when necessary — are done during construction and can be done in constant time. Therefore, if m is the total length of the N strings inserted, then this algorithm runs in $O(m)$ time.

Problem 2

Given some string S, design an efficient algorithm to find the longest repeated substring. What is the running time of your algorithm?

Solution

Algorithm: Repeat the algorithm from Problem 1, which is equivalent to finding the deepest node that has at least 2 children, except build a suffix trie from S instead of a standard trie.

Proof of Correctness: Consider the longest repeated substring of S, denoted as $s = s_1 s_2 \cdots s_l$, where each s_i is a character in the string. Because this subsequence is repeated in S, there must exist two suffixes of the form $s_1 s_2 \cdots s_l \cdots S_{end}$, where S_{end} is the last character of S. These two suffixes share the same prefix, and this shared prefix is the longest repeated substring. Thus, we have reduced this problem into Problem 1, so building a suffix trie and running the algorithm from above correctly outputs the longest repeated substring.

Runtime Analysis: Building a suffix trie can be done in $O(|S|^2)$ time (or $O(|S|)$ time if we use Ukkonen's Algorithm) and keeping track of the current longest repeated substring while constructing can be done in constant time and thus will not increase our runtime beyond this.