FIT2004 S1/2019

Assignment 3: Analysis

Nicholas Chong (29808146)

Introduction

This document will give a brief outline of my solution along with the worst-case space and time complexity of my solution. Each section will be about a single task in the assignment.

Task 1: Querying a database

The function begins by parsing the contents of the file and obtaining a list of the ID numbers (as strings) and a list of the last names of the records, in the same order as they were in the file.

We then construct a prefix trie for the ID strings and a prefix trie for the last names. What is special here is at each node (whether newly made or current), they hold an attribute called indexList. indexList contains the indices of the entries that use that specific node. For example, inserting ABC, AD, BE (the first initial node A will contain an indexList of [0,1]).

Once the trie is constructed, to retrieve the entry indices for a given prefix, we traverse down the node based on the given prefix. If we have traversed down the full length of the prefix, we retrieve all the indices in the indexList at that node. This works because the indexList of a node effectively tells us which entries use that node and so there is no need to traverse down to the leaf node in order to find all the entries.

Once we have obtained both indexLists from both tries, using their respective prefixes. We now have two lists that are sorted (because we insert entries into the trie in order of their record index) and have unique elements (because we can only add an index to an indexList at a node once). Markers are placed at the start of both lists and iterated through, adding a value to our results list if the values at both markers match. If not, we compare the values and the marker with the lower value will move to the right once. The duplicates stand for indices that match both the ID prefix and the last name prefix. We return the duplicated indices.

Worst-case Time Complexity

Iterating over every character for a record will take O(M), which is then repeated O(N) times. We then insert all the ID strings into a prefix trie, followed by all the last names into a separate trie. At its worst case, we would have to create a new node for every character. So, because we have T total characters for last names and ID strings, we would create a new node O(T) times. We also note that at each iteration, we add one and only one index to a node, this means that the worst case for adding the record indices will also be O(T) as there will be at max T iterations.

To retrieve the index list for a prefix, we need to traverse down the trie. At the worst case, we traverse down trie the same number of times as the length of the prefix. So, if the prefix was “ABCD”, we would need to traverse down the trie 4 times. To find the duplicates of two lists, one of the markers will always move to the right, or both markers will. This means that at the worst case, we would have to go through both the length of the first list and the length of the

second list. Altogether, the traversal will take O(k + l) (where k is the length of id_prefix and l is the length of last_name_prefix) and the duplicate finding will take $O(n_k + n_l)$ (where n_k is the number of records matching the id_prefix and n_l as the number of records matching the last_name_prefix). This means our queries will take $O(k + l + n_k + n_l)$.

Overall…

$$O(T) + O(T) + O(NM) + O(k + l + n_k + n_l) = \mathbf{O(T + NM + k + l + n_k + n_l).}$$

Worst-case Space Complexity

Extracting the IDs and last names will always take O(NM) as the characters will take some fraction of a record M, being repeated N times. In constructing the trie, at worst, we require one node for each character of the ID strings for the first trie and one node for each character of the last name strings for the second trie. This means that we would have O(T) nodes. We also store just one element at each iteration of building the trie, so we are also storing O(T) indices. In traversing the trie, we only hold references to the current node (O(1)) and once we reach the desired node, we return a reference to the indexList at the node O(1). References are also used for the markers in finding the duplicates. Our final list is based off a subset of the elements of the first list (size n_k) and the elements of the second list (size n_l).

Overall…

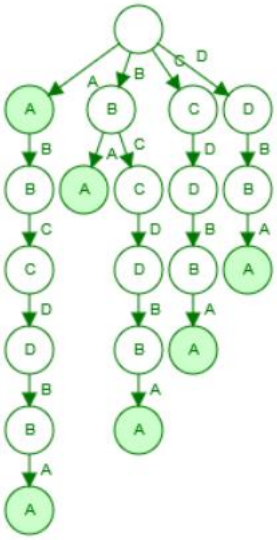
$$O(NM) + O(T) + O(T) + O(2) + O(n_k + n_l) = \mathbf{O(T + NM + n_k + n_l).}$$

<u>Task 2: Reverse substrings search</u>

This function works by initially retrieving the original word from the file. We then find every suffix of the original string. After that, we reverse the original string and find all suffixes of that reversed string. Finally, we insert all the normal suffixes into the same trie as in Task 1, forming a suffix trie. The index list (used in Task 1) now serves as the initial index to slice the string to get the suffix (e.g. the original word “apple” would have the suffix “apple” as 1, suffix “pple” as 2 and so on).

To retrieve the reversible substrings, we iterate and pass (not insert) each of the suffixes of the reversed string through the trie. If any prefix of the reversed string matches with the nodes going down the trie, that is a reversible substring (what we want). So, we keep traversing down the trie and at each node where the prefix length is more than 2, we save the prefix of this reversible string and the index at which the original suffix was inserted.This continues until the reversed string does not find a child node to traverse into. In order to avoid duplicates, when a node’s results are collected, we set the node’s isMarked attribute to true. In the rest of the strings, the results are not collected if it sees that a node’s isMarked is true. We repeat this procedure for all the reversed strings and return the collected results.

To further explain why this works, if a given substring has its reversed form in the original string, some suffix of the reversed string should be able to traverse down the trie, effectively tracing the given substring. For example, if the original word was “ABCDBA”, the initial suffix trie would be as follows.



We have our list of suffixes of the reversed string (ABDCBA, BDCBA, DCBA, CBA, BA). When we place 'ABDCBA' into the trie, AB will match, and the 'B' node will contain the index list of 0. As a result, we get ['AB', 0]. Because there is no child node 'D', we cannot traverse down further. Another example is 'BA' (the last element of that list), it will match against BA in the trie and so ['BA', 4] will be returned. For this example, the overall result is [['AB', 0], ['BA',4]].

Worst-case Time Complexity

Extracting the single line from the file will involve iterating over every character in the file which will take O(K) time. Getting the reversed form of the original string will also take O(K) as it would iterate over every character. Collecting the suffixes of the line will involve string splicing a segment of the string before splicing one character more to the right, which would take O(K) time. This is the same for getting the suffixes of the reversed string, taking O(K) time. The insertion of all suffixes into the trie will take $O(K^2)$ time as for each of the K suffixes, it would have to create a node and traverse down the trie a total of K times. Finally, each reversed suffix traverses down the trie and at each node encountered that, in total, forms a substring of the reversed suffix, we loop over the word and index list at that node. This occurs a total of O(P) times as the traversal ends when the word cannot be traversed down further.

Overall…

$$O(K) + O(K) + O(K^2) + O(P) = \mathbf{O(K^2 + P).}$$

Worst-case Space Complexity

Containing the file's contents would take O(K) space, having to hold each character in the file. Creating the reversed string will also take O(K) space as it contains the same number of characters as the original string. Creating their relative suffixes will take $O(K^2)$ space each as each line will take O(K) space and it will be repeated K times. Constructing the suffix trie based on the suffixes will take $O(K^2)$ space in the worst-case, as every character will need a separate node for itself and there are a total of K^2 characters in a list of suffixes for a word. Containing the total list for all reversible substrings will take O(P) space, as it will contain all the substrings whose reverse appears in the list.

Overall…

$O(K) + O(K) + O(K) + O(K^2) + O(K^2) + O(P) =$ **$O(K^2 + P)$.**

END OF ASSIGNMENT

THANK YOU