DAA432C
ASSIGNMENT 3

WRITE AN EFFICIENT ALGORITHM TO GENERATE AN (M X M) MATRIX FILLED WITH RANDOMLY GENERATED ENGLISH ALPHABET CHARACTERS. FIND OUT ALL VERTICAL LOCATIONS CONTAINING VALID ENGLISH WORDS. DO THE NECESSARY EXPERIMENTATION AND ANALYSIS WITH YOUR ALGORITHM

PREPARED BY

SHELDON TAURO
ASWIN VB
AVINASH YADAV
NISTALA VENKATA SHARMA

Indian Institute Of Information Technology
Allahabad

2018

Write an efficient algorithm to generate an (m x m) matrix filled with randomly generated English alphabet characters. Find out all vertical locations containing valid English words. Do the necessary experimentation and analysis with your algorithm

Sheldon Tauro*, Aswin VB†, Avinash Yadav‡ and N.V.K. Sharma§
Indian Institute of Information Technology,Allahabad
Allahabad-211012
Email: *iit2016137@iiita.ac.in, †iit2016106@iiita.ac.in, ‡itm2016004@iiita.ac.in, §ism2016005@iiita.ac.in

I. Introduction and literature survey

For finding a word in the English dictionary that is if that word exists in the English dictionary or not, we can have many searching algorithms.Of all those we have implemented data structure 'trie' for searching which is efficient of all.

The question is to identify all the correct English word from a m x m matrix having random alphabets in it. Every word of length from 1 to maximum can be created and then it is checked to be a correct english word.The m x m matrix of random English alphabets is created using random generator and then searching of the words is done. Checking of a word is done by using Trie Data Structure which is specially used to handle dictionary kind of languaage structure. The whole word is traversed and searched if it is present in the dictionary i.e. in our trie data structure.Every possible combination is first generated and then checked if it is valid or not.

In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search treean ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest. For the space-optimized presentation of prefix tree, see compact prefix tree.

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to M * log N, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements.

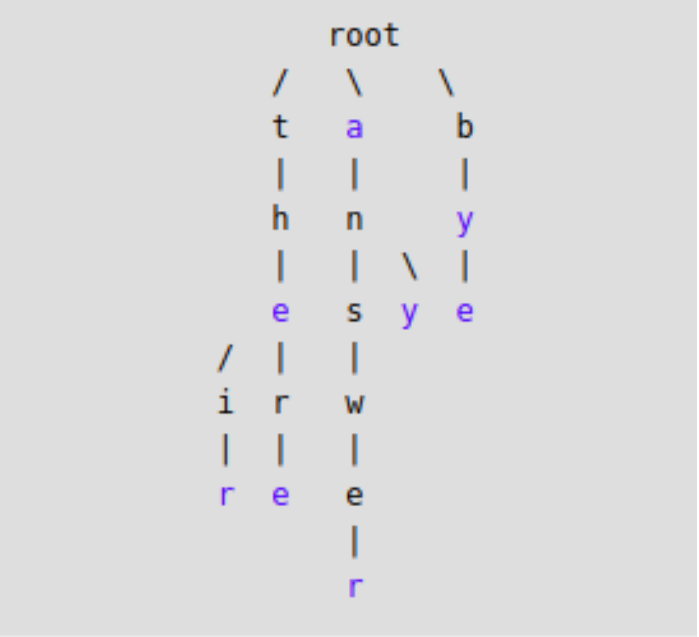

Fig. 1.

II. Algorithm Design

$temp$ and $root$ are data type of node which contains an array of node type of size 26 and a integer $isleaf$, str is an array of string (mxm) , tmp is a variable of string type, n is the number of strings defined in dictionary, s is a string in dictionary.
$Note$: – In the algorithm we have used $varj$, $index$, m as global variables.Also we have assumed the length of the longest string is unknown.

Explanation

Algorithm 1 Initializing a new node for the trie

```
function getnode()
temp ← memory
for i ⇐ 0 to 25 do
    (temp → child[i]) ← 0
end for
temp → isLeaf ← 0
return temp
```

Algorithm 2 Insert a word in the trie

```
function add(root, string s)
temp = root
for i ← 0 to (string.length-1) do
    ind ← s[i] −′ a′
    if temp.child[ind] == NULL then
        temp.child[ind] ← getnode()
        temp.child[ind].isleaf ← 0
    end if
    temp ← temp.child[ind]
end for
temp → isLeaf ← 1
```

Algorithm 3 Check if a word or its prefixes are in the dictionary

```
function find(root, string s, int index)
temp ← root
tempI ← index
for i ← index to (m) do
    ind ← s[i] −′ a′
    if temp → isLeaf == 0 then
        print answer
    end if
    if temp.child[ind] == NULL then
        return 0
    end if
    index ← index + 1
    temp ← temp.child[ind]
end for
if temp → isLeaf == 0 then
    print answer
end if
return 1
```

Algorithm 4 Main function

```
function
root ← NULL
root ← getnode()
for i ← 1 to n do
    add(root, s)
end for
for i ← 0 to m do
    for j ← 0 to m do
        tmp.push(str[j][i])
    end for
    for j ← 0 to m do
        find(root, tmp, j)
    end for
end for
```

We have nodes which stores a character,a variable which tells whether it is a leaf node (1 if it is a leaf node and 0 if it is not) and a pointer to the next node (NULL if it is a leaf node). When the dictionary is added , for each word in the dictionary a node will be created for the root node. After that for each letter a node will be created as a child which goes on until it reaches the end of the word after which the pointer of the last node is set to NULL and $isleaf$ variable is set to denote that it is a leaf node. So when we traverse through columns of the matrix each and every possible word is passed for checking and for a word, After traversing through out the length of the word through the trie if the last node is a leaf node (i.e. $isleaf == 1$) then the word is a valid word and the word is outputted.

III. Analysis

m is the dimension of the matrix. *we* have three functions find() , getnode() , Add(). In which Add() adds the dictionary data set into the trie tree and find() checks whether the given string is a valid word or not and getnode() creates a node.

- $time_{getnode}$
  $\propto (26) * (3) + 2 \propto 76$
- $time_{add}$
  $\propto StringSize * (7 + getnode) + 2$
- $time_{dictionary}$
  $\propto 1 + n * 4 + n * add + getnode$
  $\propto 1 + n * 6 + NumChar * (7 + getnode) + getnode$
  $\propto 77 + n * 6 + (NumChar * 83)$
- $time_{FindWorst}$
  $\propto StringSize * 8 + 3$
- $time_{FindBest}$
  $\propto 7$
- $time_{Query}$
  $\propto m * (4 + 4m + m(4 + find)))$
- $time_{QueryWorst}$
  $\propto m * (4 + 8m + m(\sum_{i=0}^{m}(find(i)))$
  $\propto m * (4 + 8m + 3m + m(\sum_{i=0}^{m}(8i)))$
  $\propto 4m^3 + 15m^2 + 4m$
- $time_{QueryBest}$
  $\propto m * (4 + 8m + 7m)$
  $\propto 15m^2 + 4m$

A. Time Analysis

$$f(n) = time_{Dictionary} + time_{QueryWorst}$$
$$=$$
$$(NumChar * 83) + 77 + 6n + 18m^2 + 5m + 4m^3 + 15m^2 + 4m$$
$$= 4m^3 + 33m^2 + 9m + 6n + 77 + (NumChar * 83)$$

$$g(n) = time_{Dictionary} + time_{QueryBest}$$
$$=$$
$$(NumChar * 83) + 77 + n * 6 + 18m^2 + 5m + 15m^2 + 4m$$
$$= 33m^2 + 9m + 6n + 77 + (NumChar * 83)$$

$$\Omega(g(n)) = \Omega(m^2 + (NumChar * 26))$$

$$O(f(n)) = O(m^3 + (NumChar * 26))$$

$$AverageCase = \Theta(m^3 + (NumChar * 26))$$

- The above equation of time analysis shows that when best case is calculated the time complexity is of the order of m^2 and in the worst case it is $m^3 + (NumChar * 26)$.

IV. Experimental Study

$$TimeComplexity$$

Then Time Complexity is $O(m^3 + (NumChar * 26))$, where n is the size of the array. In the best case it will be $\Omega(m^2 + (NumChar * 26))$.

- When we plot for m vs t It can be noted that, the graph is cubic for average and worst case and parabola for best case as in fig(1).The green line is the graph for best case,violet for average case and the blue line is for best case. In both the case the graph is cubic.the change in the number of steps in both cases is because when the string length is 1 then at first iteration itself it reaches the null pointer and gets out of the loop.

n	Best	Worst	Average
25	134200812	134419562	261890000
50	134229037	135979037	263373000
75	134276012	140182262	267369000
100	134341737	148341737	275124000

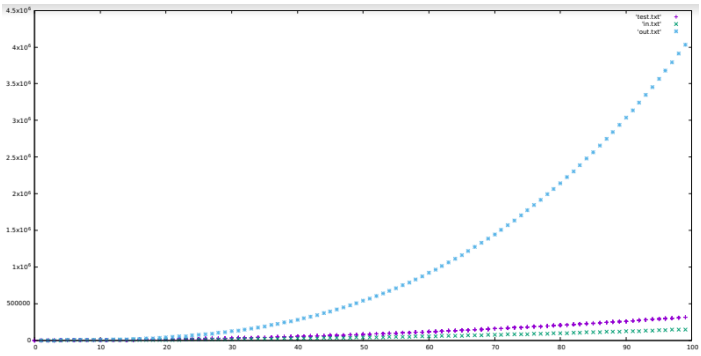

Fig. 2. Complexity Graph between m vs time.

V. Discussions

- In this algorithm we used trie tree to solve the problem which has a time complexity O($m^3 + (NumChar * 26)$).
- Another approach to solve this problem is to use binary search on the dictionary file which includes file-handling of the dictionary data set and then applying binary search on it which has a time complexity O($m^4(logm)$)
- In this algorithm if we are given a string of length n then we can search it in the trie tree with a time complexity O($m^3 + (NumChar * 26)$) in the worst case and Ω(m$^2 + (NumChar * 26)$) in the best case.
- On comparing the algorithm we found out that trie tree is a better algorithm.

Time complexity of common data structures	
Data Structure	Time Complexity
Trie Tree	O(m^3)
Binary Search	O($m^4(logm)$)

From the above table we reach to the result that Trie Tree is best data structure to solve this problem.

VI. Conclusion

The problem was to find an efficient algorithm to generate an m*m matrix filled with randomly generated english alphabets and find out all vertical locations containing valid english words and to do the necessary experimentation and analysis on the algorithm.So we implemented this problem using a trie in which a tree is made for all possible words in the dictionary, when we search vertically across the matrix we compare all possible words in the matrix with the words that is present in the trie and if there is a match it is noted down and the process is repeated for all columns of the matrix. On plotting the graph for this algorithm we get a graph of cubic equation.

References

[1] IDAA CLass Lectures.

[2] https://www.geeksforgeeks.org/trie-insert-and-search/

[3] https://en.wikipedia.org/wiki/Trie