Final Project Report

109000168 許媄香

* Code Implementation

Text

Description automatically generatedText

Description automatically generatedI implement trie as the basic of the data structure. So, I create one class (for the tree) and one struct (for the tree’s node) for the trie data structure.

I use a pointer array with size 26 (alphabet numbers) and a Boolean to check if that letter is the end of the word for each node.

Text

Description automatically generatedBecause trie is a prefix tree, and in this project, we also need to search suffix word, therefore I create two root nodes in the trie class, one for prefix and the other for suffix tree (reverse character from the string). Also, when new node was created, I assign all of the 26 pointer point to null and the boolean “**isEndofWord**” to false. And when creating a trie of a string, I start it from the root of trie (I assign a copycat of it since root pointer shouldn’t be changed) and search the index (alphabet index start from 0 to 25) and check if that member of the node pointing to null or another node. If it’s null then create a new node otherwise just go though the direction which point by the node pointer. Just go through that until the end of string. After the position is there (end of string), assign the boolen “**isEndofWord**” to true since it’s the end of the word.

Thanks to the template for reading the correct string format from TA, I can easily input the string form the article. I create each trie for each article and store the title inside trie class. To store the trie, I create a vector of trie and push back the trie when the new one is created. After that, I just input the string inside the trie until the end of article. Besides, I increase an integer data type “**DATA\_SIZE**” by one when creating a new trie to keep in track how many articles are in the data folder in order to print out the title for the output.

After the data (article) input finished, then go to the next step, the query. Here we need to read it line by line, therefore I use getline for it. After that, I split it word by word. I add additional int “**op**” to check if there’s operator from the previous input or not and a boolean data type “**match**” with size of data input (article) to check if the query’s string matches the words in article or not. I assign “**op**” to 0 and “**match**” to false in every query line. Also, a **found** boolean to check there’s any article match to the string or not (initial value also false in every query line).

So, I use a for loop of each line of query and for each word of the query I check it in all of the article first and update the “**op**” value to 1 if it’s “**+**”and 2 if it’s “**/**” and also update the “**match**” value to true if the string match inside the article. And before checking the string from trie, I make an if statement when “**op**” is not ‘0’ meaning that the previous string or word is an operator, therefore I check if the previous word before the operator matched the article (in each for loop) or not. If the operator is **AND** (**+**), also the previous word doesn’t match in the article, then I just skip it and go to next article, but it matches, then I need to check it according to the string format (prefix, suffix or exact). Same for **OR** (**/**) operator but the contras, if previous word match in the article, I’ll just go to next article and if it doesn’t then check it according to the string format and update the **match** value from the search function return value.

For the search function, there are three kinds of searching, prefix, suffix and exact word. And the searching is the same as create a tree, the only different is if the node is pointing to null I just return the function to false and if the loop finished without any problem then return true (if it’s not exact word searching). In prefix and exact word, I use prefix trie to check if it’s match (the child of the char index not pointing to null). The only different for exact word is it return “**isEndofWord**” value of the node. Suffix is the same as prefix but with different trie root.

After each line of query, I insert the article title to output file if it’s match otherwise, I insert “Not found” string there.

* Challenge

The challenge is of course to make the code run faster. After finished my code, I tried to find out which part of my code is doing an overlapped thing. And I found out a lot, but the time decreasing part is the query part (there’s **+** as **AND** operator and **/** as **OR** operator). For **OR** operator, I forgot that if one of the operands is is ‘1’ or true, then the answer should be true, therefore there is no meaning to check the other operands it it’s ‘1’ or ‘0’ value. And the same for **AND**, it one of it is ‘0’ then I don’t need to search for the next string if it’s matched the string in the article or not.

* Reference

<https://www.geeksforgeeks.org/trie-insert-and-search>