Autocomplete

Sergi Martínez

May 2019

1 Introduction

In this document can be found a brief report of my implementation for the Autocomplete algorithm. It contains an explanation for each of the classes, data structures and algorithms used.

2 Trie and Nodie

These both classes form the main data structure used for the resolution of this problem. The prefix trie, AKA Trie (as it will be referred in this assessment). There is a Wikipedia article [1] where is perfectly explained, and features a code written in pseudo code which helped me during the implementation of the class. Basically I split the Trie in two parts: the trie itself and the nodes (called Nodie), so each part is a different class.

Nodie (the node) consists of a character, a HashMap ⟨Character, Nodie⟩, and a boolean that tells if the current Nodie is a leaf or not. Since a trie is supposed to work just with letters of the alphabet, the recommended implementation uses an array of characters instead of a HashMap. This is due that you can access easily to the characters if you know the alphabet index(like the key of a HashMap). But I noticed one of the Keywords contains a special character, so this array of characters cannot be used. Instead, I use this HashMap ⟨Character, Nodie ⟩which also works great for the purpose.

The Trie class is where all the main computations are made. The class attributes is just a Nodie, which represents the root of the Trie. (Note that you can access all elements of the Trie just by having the root). Regarding the functions, Trie contains 3:

- **insert(String str)** As the function name indicates, it basically receives a String **str** and inserts it into the Trie data structure.
- **getSuggestions(String str)** This is a void function that calls a recursive function later on . It receives a String, and iterates over it so it goes depth into the Trie searching for the Nodie that corresponds to the given String **str**. If it doesn't find any node, it does nothing. (the progam couldn't find any suggestion) . Once the Nodie is found, it will call the recursive function. (See below). After the recursive call have finished, there will be an ArrayList containing all the words it could find. So it's time to iterate over it and print the results. The assessment asks for only 4 suggestions, so it will print maximum 4 sorted suggestions. In order to sort the ArrayList, I had to call the sorting algorithm provided by Collections. This may be a little bit inefficient, but I believe it's the only way to do it. Keep in mind that we can't assure that the keys in the nodes' HashMaps will be sorted.
- **recursiveCall(Nodie nod, String str, String c, ArrayList ⟨String⟩suggestions)** This is the recursive function which finds all the suggested keywords from the input string entered by the user. In tree data structures, recursion is used to find all the children nodes of a given node. So this is what we need in order to find the suggestions. The function obtains a String **c** which will be used to store recursively the suggestion. Once the function reaches the base case (it finds a leaf), **str** string will be concatenated to **c** to create the final found suggestion. Then it will be stored in the ArrayList **suggestions**.

I implemented the insert function with the help of a pseudo code provided at the wikipedia article. For the other functions I had to improvise a little, since the implementation of the Trie and Nodie classes are different.

3 Autocomplete

Here is where the main function is found. It just reads the input.txt file and stores its values into a Trie data structure. It asks then the user for words to search. A maximum of four suggestions are given to the user for each word entered.

4 Optional exercises

1. As far as my knowledge of computing algorithms goes, a prefix tree is the best option for solving this kind of problems. I believe now it's a matter of how well optimised the Trie is. It would be necessary then to sort the Keyword list before inserting it into the Trie. If the input Keywords is sorted, then the binary tree will be so, and the search for the suggestions will be faster. So it's a matter of how to Trie is built.

2. It will be necessary to create a new Trie and store there the same words as in the first Trie, but backwards. Then the program should iterate over the input string starting from the middle, going to both ways and calling the respective function for each direction.

References

[1] Trie wikipedia. `https://en.wikipedia.org/wiki/Trie`.