T9 Algorithm

Aditya Gupta

October 2022

1 Introduction

In this assignment, we implement the T9 algorithm, which was used in old mobile phones for auto-correction. This algorithm makes use of the tries data structure which is used for storing a collection of strings and making efficient search operations on them. This data structure is also referred to as a prefix tree as this data structure is used for prefix-based searching, unlike a hash table.

2 User input based on the old keypads

For this assignment we will take input in the form of character strings that comprise of sequence of numbers that correspond to certain numbers being pressed on a keypad. The keys in old phones mapped multiple characters to the same keys.

Keyboard Layout (Based on Swedish mobile phone keyboards):

- 1: a,b,c
- 2: d,e,f
- 3: g,h,i
- 4: j,k,l
- 5: m,n,o
- 6: p,q,r
- 7: s,t,u
- 8: v,w,x
- 9: y,z,å
- 0: ä,ö," "

Now in this assignment we try to simulate this keypad experience in the form of the character input the user gives us. We try and recreate how the tries data strcuture was used on those old phones. For example, if the user typed ”253”, they might possibly want to write something like ”dog” unlike something like ”eoi”. So using this algorithm, we try to predict what word the user is looking for from the sequence provided to us. Thus in a way try to auto predict all possible words that the user might be trying to type.

3 Nodes of the Prefix Tree

The nodes for this data structure contain the following attributes :

- endOfWord: This is a boolean variable that we will use to check for the validity of words.
- array: An array of 30 Nodes (for all words in the Swedish alphabet) that is used to store all the possible branches of characters that continue on from this word.
- character: This is used to store the character of this specific node.

```
public class Node
{
    private Node[] array;
    public boolean endOfWord;
    public char character;
    public Node(char c)
    {
        array = new Node[30];
        endOfWord = false;
        character = c;
    }
    public Node(){
        array = new Node[30];
        endOfWord = false;
    }
}
```

4 Adding Words

We can populate the trie, by starting at the root and then going through the word character by character and populate all the branches in the tree. We check if the character is valid or not by using the add() function and adding a character to the node if it’s valid. A character is considered valid as long as it is one of the characters from the Swedish alphabets in other words if it corresponds to any of the keys on the keyboard. After this we continue to do this process for the node where we placed the current character.

```
public void addWord(String word){
    Node current = root;
    char c;
    for (int i = 0; i < word.length() ; i++) {
        c = word.charAt(i);
        if(current.add(c)) current = current.get(c);
    }
    current.endOfWord = true;
}
public boolean add(char c)
{
    int index = calcIndex(c);
    if (index < 0 || index > 29) return false;
    if (array[index] != null) return true;
    array[index] = new Node(c);
    return true;
}
```

5 Searching for a word

First we take the character sequence taken from the user and make a 2d array based on it which contains all possible characters that might be part of the possible words that map to the character sequence. For example if the user input is given by "1234" we should get a 2d character array $\{\{a,b,c\},\{d,e,f\},\{g,h,i\},\{j,k,l\}\}$. We have a function called searchWord() that does the above for us. We store the the possible characters in a 2d array called outcomes. Once we have have filled up the outcomes array we call the prefixSearch() function to search for the words that are related to the character sequence passed by the user as input.

After the prefixSearch() function is called we recursively search through the prefix tree or trie. We iterate through the tree recursively till we reach the last set of characters upon which we look for the nodes that are not null. From the non null nodes we can find the list of possible words that correspond to the character sequence provided to us by the user. After this we go through all the possible words through iteration and print them out to the console. We pass in 4 parameters as a means of keeping track of our current position inside of the tries data structure. We have the outcomes array that we created in the searchWord() function, followed by the index, node we are currently on and finally the word that we are building using this data structure.

```
private void prefixSearch(char[][] outcomes, int index, Node node, String word)
{
    int length = outcomes.length;
    if (index < length)
    {
        for (int i = 0; i < 3; i++)
        {
            char c = outcomes[index][i];
            var node2 = node.get(c);
            if (node2 != null)
            {
                String temp = word + node2.character;
                prefixSearch(outcomes, index + 1, node2, temp);
            }
        }
    }
    else
    {
        if(node.endOfWord) System.out.println(word);
        for (int i = 0; i < 30; i++)
        {
            var node2 = node.get(i);
            if (node2 != null)
            {
                String temp = word + node2.character;
                prefixSearch(outcomes, index + 1, node2, temp);
            }
        }
    }
}
```

We use the above functions to go through the text file provided for this task which contained the most common words in Swedish text. To test the proficiency of the T9 algorithm I added all the words from the kelly.txt file into the tries data structure. After this I searched for all possible words that correspond to the character sequence ”123” in the text file. This gave me a list of words that correspond to pressing the first 3 keys on an old mobile phone. After this I did a few more tests to check if this algorithm worked for other character sequences and it worked perfectly.