Trees - Breadth-first Search

Daniel Sherif

Fall 2024

Introduction

In the following report, we will create and walk through the implementation of a T9 keypad system. The T9 works by converting inputs from a nine digit keyboard and predict the outcome in a way. T9 keeps track of all words that can be encoded and narrows down possible words based on amount of inputs. Though, the T9 system will need to be "trained" in a sense. This implementation will use Swedish words and the letters "åäö" also.

Using a Trie

We start with the core structure that all methods base themselves of, the Trie. The trie is basically a tree, but its meaning depends on the path that the nodes take. So for example. By combining linked lists, trees and arrays, we get the ability to assign each node 27 different slots, for the letters a-ö, to branch off. This makes it possible to account for every word in the dictionary, by taking the proper path in the trie.

So if we were to spell "harvard", we would start from root which is empty, then branch out of the slot index which accounts for the character h. From there we go branch out of a until we reach d, the last character of the word. From there, we have successfully accounted for all letters, so we set a boolean inside the node that accounts for d to true, confirming that there is a word, with that length, that ends in d and takes the same path. See Figure 1 for initialization of the Node, to get an idea.

```
public Node() {
    next = new Node[27];
     valid = false;
     // indicates if the path from root is valid
}
```

Figure 1: *Node method inside node class.*

Populating the Tree

To add the words, we use a file filled with words, that get decoded and added to a ArrayList. We iterate through the trie, if we take the "harvard" example again, we start from h, add this to `node.next[index of char h]` and iterate like that until we reach the end of the string. If next is empty, we create a new node that accounts for it in the index of character h. See Figure 2.

```
...
    for (int i = 0; i < word.length(); i++) {
        c = word.charAt(i);
        idx = code(c);

        if (curr.next[idx] == null) { // if no branch, construct
            curr.next[idx] = new Node();
        }
        curr = curr.next[idx]; // traverse;
    }
    curr.valid = true; // mark end
...
```

Figure 2: *How to add words to trie, inside method "addWord(String word)".*

There is a "mother" function to this as well called add which does Figure 2 iterated for every word inside a given file. We omit showing this, as it is not too relevant.

Notice how, if there is no branch, if next equals null, we construct a new node there, then traverses. And if there is one, we do nothing and just traverse through it normally. Then when we finally exit the for loop, which is the word length of the given word, we set the last nodes attribute called "valid" to true, which indicates that there is indeed a word path leading to this node.

Finding words from the Tree

The searching method is much more tricky then the rest. Not so much in complication but more in initially implementing it, more on this in the conclusion section of the report.

Like previously, the finding of words has two functions, one "mother" function called decode and one larger one that is recursive called connect. Decode takes a certain "key", which signifies the written "code" by the user of the

T9 system, and turns that into possible words. Decode also creates an array list which it puts all these words into and returns. It turns these codes into words by dumping a bunch of information into the collect function, which is void and changes the ArrayList by adding to it iteratively through recursion. See Figure 3.

```
private void collect(Node curr, String word,
String key, ArrayList<String> words) {
    if (!key.isEmpty()) {
        char c = key.charAt(0);
        int n = keyToIndex(c);

        String newKey = key.substring(1);
        // All characters without most recent

        for (int i = 0; i < 3; i++) {
            if (curr.next[n * 3 + i] != null) {
                c = reverseCode(n * 3 + i); // get char
                collect(curr.next[n * 3 + i], word + c, newKey, words);
            }
        }
    }
    if (curr.valid) {
        words.add(word);
    }
    return;
}
```

Figure 3: *The recursive collect method, that adds words to a ArrayList.*

Results

Testing of the decode method, which in turn sufficiently tests practically all other methods, was done by putting different keys into the decode method and printing the lists they return. Some individual test to methods like addWord was also done through breakpoints and such, this will not be highlighted here. The results obtained from the keys 15261, 11462 and 752224 are as follows.

```
Decoding '15261':
Possible words:  [andra, an, bo]

Decoding '11462':
Possible words:  [bakre, bak, bal]

Decoding '752224', ex from assignment:
Possible words:  [toffel]
```

Results 15261 and 11462 were derived from the words "andra" and "bakre". The last number, "752224", is derived from the word "toffel". This is ofcourse apparent in the results but they were gotten through a method called charToKey, which iteratively turned every character to a key press on the T9, which allows for better testing.

Conclusion

I had two large problems when implementing this T9 keypad. First was getting a text file with "åäö" letters to read properly in java. I believe that I downloaded java in a weird way or that my PC is setup in a way where preferred or standard language is English. I am not certain but after hours of scouring the internet, I finally found a way. I just needed to add the charset name, UTF-8, as a parameter to the reading object, but this needed different object types then I initially used. See Figure 4.

```
...
    try (BufferedReader br = new BufferedReader(new
    InputStreamReader(new FileInputStream(file), "UTF-8")))
...
```

Figure 4: *File reading implementation that works with UTF-8.*

Problem that came up shortly after was with the decoding of words. Bascially, my implementation was very close to correct but relied on way to many variables. After, again, some extensive scouring of the internet, chatgpt stepped in with some tips. Basically it advised to use substrings and abuse the fact that I was using strings to my advantage, which I ended up doing. So shout out to GPT for that one, real life saver. It taking me so much time is also a big reason why I just included the whole method into the report, which I always try to avoid, I am just too proud of it and have to display it.