Algorithm for Morse Code Problem without Silences

Deepak Gautam
deepugtm@gmail.com

Table of Contents

1. Algorithm Background

Because the vehicle of the Captain Smith is in trouble, he has not enough time for all the silences between symbols and words of the Morse Code. Hence the code Smith sent will have various possible meanings, as there is no unique solution to the sent codes without silences in between words and symbols. The algorithms' task is to find all the possible sequences of letters and numbers in efficient way.

2.Methodology

The algorithm has following steps specifically.

2.1 Setup

In this step, algorithm initializes the maps of Morse Code and Reverse Morse Code(for verification of the result output sequences). Morse Code map is a HashMap with keys as a Morse-Code values and value is a one of the valid alphabetical character or digit character. Hence this map will have 36(26+10) keys. The other map is the reverse of this Map, which will have same number of keys but the keys will be one of those values in Morse Code map.

2.2 Building of Matrix

For every position of a character in the Morse Code input string, a substring up to four next characters is calculated. For each position, this list of substrings then checked with Morse Code map if such substrings maps to any character value from Morse Code map. This gives a string of possible characters for the given position.

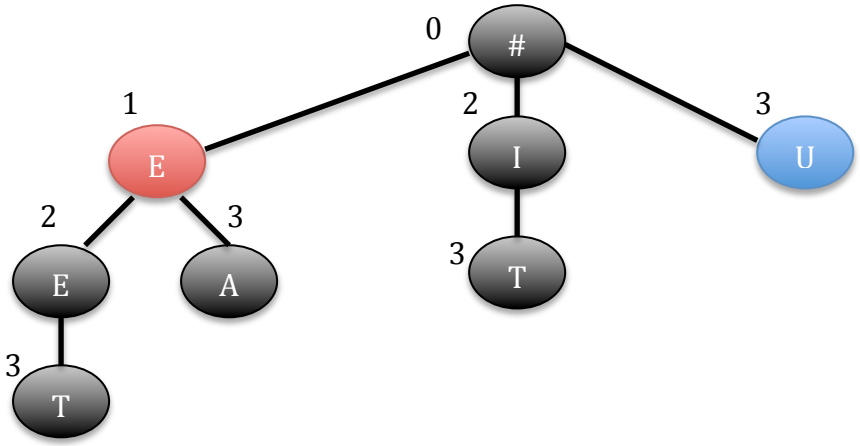
For Example, for input string 'OOA', these are the possible characters for first position.

- If only first dOt is taken into account, output alphabet should be 'E' (for single dot)
- If first two dOts are taken into account, output alphabet should be 'I' (for double dots)
- If all three characters are taken into account, output alphabet should be 'U'.

In this way, the string for the first position is "EIU". Similarly there would be *n* such strings for every position *i,* up to n. List of such strings is called matrix of the possible characters in the output string. If we give the indices, *i*, for the matrix entries starting from 0, *item[i]* represents the possible characters for the i^{th} position. e.g. item[0] = "EIU", item[1]="EA" and so on.

2.3 Building of Tree

The matrix built in above step (2.2) is used to construct an n-ary tree in such a way that if we list out all the root-to-leaf paths of the tree, we get all the possible sequences of the alphabets and numbers for the given input Morse Code. The weight of a node is sum of the weight of its parent (which is 0 for the root #) and the number of Morse symbols when it is represented in Morse Code.



The tree shown above is for Morse Code input 'OOA'. Root node has zero weight and its value is '#', which has no impact/meaning on output. It has three branches E, I and U. This is identical to matrix entry item[0]. E is 'O', I is 'OO' and U is 'OOA' in Morse code, hence their weights are 1, 2, 3 respectively. The children 'E' and 'A' of the node E on the left (marked red) have weight 2 and 3

respectively summing up their individual values with parents' weight. This is no coincidence that these two nodes have same character values 'E' and 'A' as characters from item[1]. This is where the matrix is used to construct the tree. The characters of the item from the matrix become the children of a node. The index for the matrix from which to choose the item for building children nodes is determined by the weight of the node itself. For example, weight of the root node is 0, hence the desired index is 0. Which means, the children of the root node should be characters from item[0]. This process of filling tree nodes goes on until the weight of the node equals the length of the input string of Morse Code. In above example, the length of input string is 3 and the weight of the node 'U' on the right side (marked blue) is also 3. This is why the node 'U' has no children and it is the leaf of the tree.

In General,

Children => item [weight of the node] for weight<length (input)

The weight of the node plays important role in branching the tree with the possible values filled in the matrix. Thus constructed tree has several paths from root to leaves. Such paths give the desired sequences of numbers and alphabets.

2.4 Building Output

Thus constructed tree is passed to output generator, which basically uses a recursive traversal algorithm from root to all leaves. Such paths give all the required output strings.

3.Algorithm

- Initialize 'Morse Code to Character' maps- X_CHAR (*Codes.java*)
    - ONE_CAR<String, Character>
    - TWO_CHAR<String, Character>
    - THREE_CHAR<String, Character>
    - FOUR_CHAR<String, Character>
    - FIVE_CHAR<String, Character>
- Initialize 'Character to Morse Code' map- M_CODE<Character, String> (*Alphabets.java*)
- Build Matrix
    - matrix = new ArrayList<String>()
    - for i = 0 to sequence.length()
        - possibles = empty
        - for j = 1 to 5
            - for valid last_index = i + j
                - substr = sequence.substr(i, last_index)
                - if substr is in one of X_CHAR,
                    - c = X_CHAR.get(substr)
                - possibles.append(c)
        - end-for
        - matrix.add(possibles)

- end-for

- Build Tree
  - buildTree(node)
    - if (node.weight() >= matrix.size()) return
    - children= matrix.get(node.weight())
    - for child in children
      - newNode (child)
      - newNode.setWeight(weight + getweight(child))
      - node.addChild(newNode)
      - buildTree(newNode)

- Build Outputs
  - traverse()
    - path = empty
    - paths = empty
    - traverse(root, path, paths)
    - return paths
  - traverse(node, path, paths)
    - path.add(node)
    - if(root.isLeaf())
      - paths.add(path)
    - else
      - for child in node
        - traverse(child, path,paths)
- Verify Outputs
  - paths= tree.traverse()
  - for path in paths
    - print path

4.Complexity

Time Complexity

There are basically four steps as described in section 2. The initialization phase (section 2.1) takes almost negligible time compared to other steps and hence can be ignored in calculation of over all complexity. In second phase (section 2.2), the matrix is built. Its complexity varies according to the length of the input

string. For every character in the input string, there are constant five iterations for determining next five characters as the Morse Code can have length up to five characters. Hence, the time complexity of this step is O(n). In the third step (section 2.3), the tree is built in 2^n iterations in worse case scenario via recursive function calls. So, it's upper bound is $O(2^n)$. Same is the case for iterating the outputs in last step (section 2.4).

Hence,

$$Overall\ complexity = O(n) + O(2^n) + O(2^n) = O(2^n)$$

Space Complexity

The initialization phase(2.1) builds six hash maps in total. Five for the five type of Morse Code to alphabet/number based on the length of the Morse Code for given alphabets. For example, single length map contains only two entries for Morse Code characters 'A' and 'O' respectively for alphabets 'T' and 'E'. The total entry in these five maps is 36. Reverse map also consists of 36 entries. This is not much overhead to the program compared to other ones. The second and third steps are the most heavy in terms of space complexity. These steps make recursive calls, and the number of recursive calls depends on the length of input string. Longer the input string, more memory is needed to save the state of the functions in each recursive calls. If these two steps are implemented without using recursion, memory consumption can be lowered.

5. Outputs

For given input string "OOOAOAOAOOOAOA", there are 9438 outputs generated in about 0.388 seconds in my Core i5 machine with 4 GB Ram and Mac OSX 10.10 Yosemite operating system. The outputs are included in the assignment.zip package with name output.txt. Other test cases are included in TestCases.java file inside 'com.dg.assignment.test' java package.