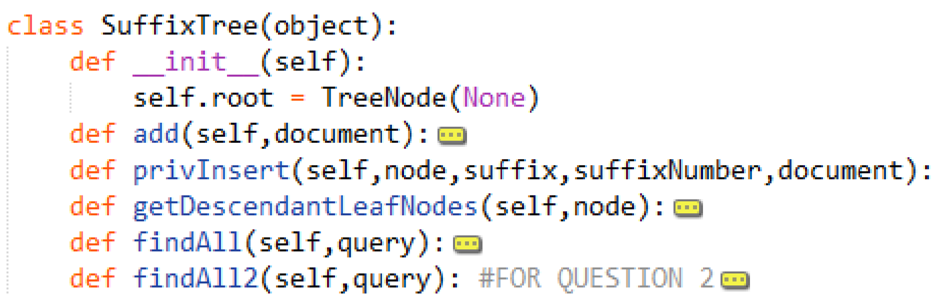
Advanced Algorithms Assignment -2

Question: Given a set of documents, build a suffix tree to perform the following operations.

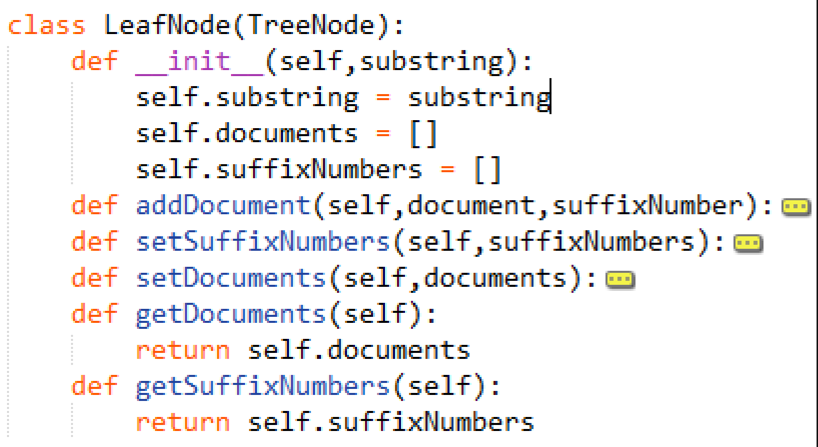
1. List all the occurrences of a query-string in the set of documents.
2. List only the first occurrence of a given query-string in every document. If the query-string is not present, find the first occurrence of the longest substring of the query-string.
3. For a given query-string (query of words), list the documents ranked by the relevance (custom relevance definition expected).

Approach: Language used: Python (ver 3.6, but data structure is version dependent only for print() statements).

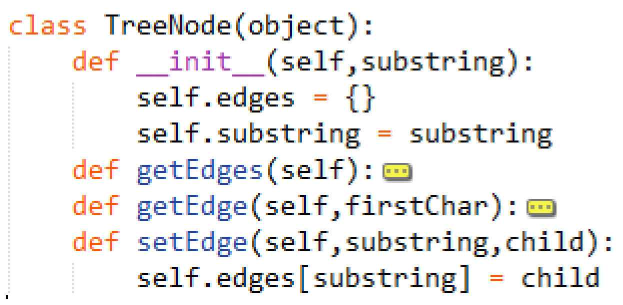
The data structure has 3 classes: (a). SuffixTree (b) TreeNode (internal node) and (c) LeafNode (the terminating node). The brief definitions of each are as mentioned:



```
class SuffixTree(object):
    def __init__(self):
        self.root = TreeNode(None)
    def add(self,document):
    def privInsert(self,node,suffix,suffixNumber,document):
    def getDescendantLeafNodes(self,node):
    def findAll(self,query):
    def findAll2(self,query): #FOR QUESTION 2
```



```
class LeafNode(TreeNode):
    def __init__(self,substring):
        self.substring = substring
        self.documents = []
        self.suffixNumbers = []
    def addDocument(self,document,suffixNumber):
    def setSuffixNumbers(self,suffixNumbers):
    def setDocuments(self,documents):
    def getDocuments(self):
        return self.documents
    def getSuffixNumbers(self):
        return self.suffixNumbers
```



```
class TreeNode(object):
    def __init__(self,substring):
        self.edges = {}
        self.substring = substring
    def getEdges(self):
    def getEdge(self,firstChar):
    def setEdge(self,substring,child):
        self.edges[substring] = child
```

Creation of Tree:

A suffix tree can be created by initializing an object of the SuffixTree class, whose constructor takes no arguments. Once initialized, a 'root' node is created (of type TreeNode class, with no 'substring' attribute). 'Substring' attribute of a node, is a string which led to that node.

Each object of type 'TreeNode' has a dictionary of 'Edges' or children, where Key is the string on the Edge and the Value is the reference to the object to which the string leads to (which can be a TreeNode or a Leaf node).

A word can then be inserted into the tree using the 'add' method, of SuffixTree class. The method used is a close implementation of McCreight's[1] Algorithm for building Suffix Tree, but it doesn't use suffix links. The method does the following:

1. Generate all suffixes of the word and pass it onto the privInsert() method, along with the suffix number.
2. The privInsert() method then does the following for each incoming suffix:
   a. It gets the first edge which corresponds to the first character in the suffix.
   b. If no such node exists, it creates a new LeafNode object and assigns that to the Edges

dictionary of the root.
c. If such a node does exist, then it checks if it's a LeafNode or TreeNode and does the following: For LeafNode, create a new TreeNode and a new LeafNode and assign the corresponding suffixes. For TreeNode, check till a point where the string on edge and suffix match and recursively create TreeNodes/LeafNodes as per conditions.

3. The edges from a tree can then be accessed via 'root.edges' attribute of the SuffixTree object.

Note: This method of insertion works for both: Single string Suffix tree and Generalized Suffix Tree(GST)[2]. For GST insert the words one by one, by calling the 'add()' method for each word.

Complexity of Insertion:

As mentioned above, for creating GST (used for question 1), we insert the words one by one. This means, if there are 'n' words (or documents), then we do 'n^2' number of function calls. If we were to use suffix links, then this complexity would've reduced to 'n'. Thus, the complexity of this algorithm is: ***$O(n^2)$***.

Statistics of insertion:

Average insertion time, for Dataset1 (Aesoptales), for creating a single GST**: 5.7536 seconds.**

Average insertion time, for Dataset1 (Aesoptales), for creating 'n' trees, one for each document: **2.5846 seconds.**

Approach for Question 1:

The 'findAll' method of SuffixTree class handles searching for a string in the tree, along with a private method: getDescendantLeafNodes.

As name suggests, getDescendantLeafNodes returns a list of leaf nodes for any input node. If input is a LeafNode, then it returns an empty list. The list contains references to leaf nodes. It works by recursively calling the same function (getDescendantLeafNodes) until we reach a leaf, which won't have any edges outwards.

The findAll method is based on the following steps:

1. Input: a query string. Get the first child, which has the same starting character as the query string.
2. If no such child exists, then return an empty list.
3. If it does exist, then:
a. Get the length of child's substring. Compare this substring with query, until a point of mismatch.
b. At point of mismatch, see which string was exhausted. If query string was exhausted the do step (d), else do step(c).
c. At this point, get the next child, which corresponds to the first character of unmatched query and go to step (a) and continue until child either becomes None or query is exhausted.
d. If query got exhausted, then use the getDescendantLeafNodes method on current node, to get all result nodes and return them.

Statistics of searching for certain strings:

Average time for searching common string **'one'** on single GST: **0.001463 seconds**.

Average time for string "and thus addressed the Crow" on single GST: **0.0001406 seconds**.

Approach for Question 2:

For solving question 2, ‘n’ suffix trees were used, which reduced the tree building times by a third at least.

The ‘findAll2’ method was used, which again uses the same getDescendantLeafNodes for getting all leaf nodes of a node.

The ‘findAll2’ method is based on the following steps:

1. Generate all substrings of given query, which takes **$\Theta(m^2)$ time**, if ‘m’ is length of query. Sort this list of substrings in descending order of their lengths.
2. Create ‘n’ trees, where ‘n’ is number of documents.
3. Then, for each tree, we do the following:
   a. Find the first substring from substring array exists in the tree, if it does, append it to results list and break.
   b. If it doesn’t exist, then continue for every other substring, until a match occurs.
4. At any given point, number of results equal number of documents.

Statistics of searching for certain strings:

Average time for searching **‘charger’: 0.0303 seconds.**

Average time for searching **‘it would be as if I should beg every Dog’: 1.2362 seconds.**

Approach for Question 3:

Question 3 posed an interesting challenge. It required to make use of a heuristic which would order the search results according to a relevance order.

The same approach of building ‘n’ trees is used for this question too.

Given a query, the relevance order used was:

1. Highest relevance given to a tree where entire query string is present.
   Note: If query is “he is good” then the highest relevance is given for “ he is good ” (notice the leading and trailing spaces).
   This is because, “he is good” (without leading and trailing spaces) would also match “She is good”, which might not be what user is searching for.
2. Next higher relevance is given to “he is good” (without leading and trailing spaces).
3. The next levels of relevance or ranks are done as follows:
   a. Find the substrings of query string.
   b. For each tree, find the highest length query and store that query match result in a result list.
   c. Then return results list in descending order of query lengths matched.

Statistics of searching for certain strings:

Average time for searching default string **“it would be as if I should beg every Dog”**: **2.12736 seconds.**

Average time for searching custom string **“charger”: 0.06277 seconds.**

Conclusion

This report introduced an implementation for Suffix Tree Data Structure. Although this implementation is based on a naïve algorithm, it can be further improved with concept of Suffix Links. It also presented the approaches taken for the three functionalities asked.

I would like to thank our professor NS Kumar for taking his valuable time to help me in time of need, this assignment wouldn't have been completed without his help.

Future Improvements

The following changes and/or additions will be made, in coming days : First: adding suffix links to decrease complexity to ***O(n).*** To implement Ukonnen's algorithm, which is on-line, unlike McCreight's which compares every suffix from start. And to develop a better heuristic for question 3, to make the search more meaningful.

Bibiliography:

[1]: Suffix Trees and their applications, by M Maaß - 1999

[2]: Wikipedia -Generalised Suffix Tree.

Other references:

1. http://cs.au.dk/~cstorm/courses/StrAlg_f12/slides/suffix-tree-construction.pdf
2. https://www.cs.duke.edu/courses/fall14/compsci260/resources/suffix.trees.in.detail.pdf