Suffix Trees

Arun John, *Student*

***Abstract*— Searching for a word or a set of words through a document may take a lot of time if the search is not implemented correctly or an inefficient algorithm is used. Here we will be showing one of the methods to search through a document which contain Aesop's tales. Suffix trees are an efficient way to search for a substring of a string or the string itself as they take linear time for the search. Theoretically these suffix trees can also be constructed in a linear time but here we will be using an approach which takes quadratic time to construct the suffix tree.**

***Index Terms*— Aesop's Tales, Generalized Suffix Trees, Suffix Trees.**

I. Introduction

Text search through a document is a problem with many solutions. Here we will be looking at one of the ways of solving the problem of text search i.e. Suffix Trees. In particular, we will be using Generalized Suffix Trees.
According to Wikipedia, a Suffix Tree is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix Trees are used when we have a single word to search in. Here our requirement is to search for words/strings among many words, so we will be using a Generalized Suffix Tree instead of a Suffix Tree.

II. Input file characteristics

The input file being used for this can be found here, with the name AesopTales.txt. The input file contains 311 tales which follow a specific format.

Title

Story

Moral (Optional)

Each tale is ended by two newlines. These tales have a variety of words, in which the characters can belong to any case, special characters are also allowed in this input file. The title, story and the optional moral are all separated by a single newline.
For our problem of text search, we will be disregarding the case of the text and any punctuation or special characters. Also it is possible that a particular tale might have a single quote followed by a character right after it (for e.g. "Wolf's"), so we have to take care of occurrences like these. We will be considering any occurrences with a single quote and a character as a single word (i.e. "wolfs").

III. Data cleaning

After looking through the format of the input file and the way of the stories are given in the input file, we have to come up with a way to store these stories along with their titles separately from each other story. One way to do this is to put them into a list, which looks something like this.

[["Title", "Story"], ["Title", "Story"]….]

By storing it in this way, we can easily retrieve the results when we need to show the results. But before storing it in this way we first have to combine the story and the moral, separating it from the title of each story. Once this is done we have to make sure that the story and the moral together form a single string instead of multiple strings.
In our implementation of a Generalized Suffix Tree we will be using only lower-case words, instead of taking care of both cases. We will also not be taking care of special characters in our Generalized Suffix Trees, so we will have to clean that part of the data out from the stories too.

The methods described above are implemented in the **data_clean.py** file, in the **get_data()** method. The file also has a **verify_data()** method which checks if the data returned is in the particular format defined.

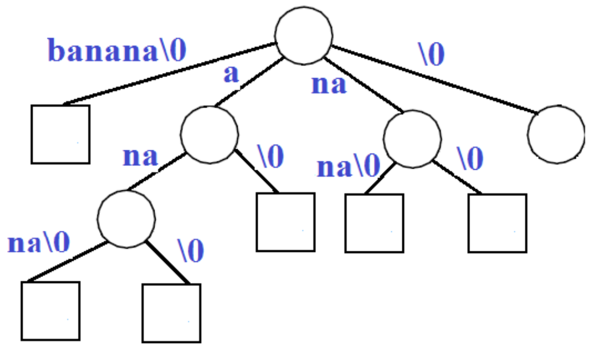
Note: get_data() may have undefined behavior when the input file is changed or is full of unknown formatting. verify_data() may also have undefined behavior if get_data() shows undefined behavior as it checks if the data is in the format given above. One should use these methods carefully as they might give undefined behavior with other data.

IV. Suffix tree creation

Once data cleaning has been done, the suffix tree has to be built from the clean data. Here we will be using a GStree class which has methods for the different problems we are going to solve with the Generalized Suffix Tree.
Before going forward and implementing the Suffix Tree, we have to define a structure for the nodes that we are going to use for the suffix tree and then make the suffix tree.

A. Node Structure

Determining the node structure is an easy task. Just by looking at a graphical representation of a suffix tree can give us a good idea of what a node should contain. Let us look at a suffix tree to make it more clear.



As we can see above, the root node is like the internal nodes, only having links and no data/indexes of its own. For simplicity, we will use the same type of nodes for both the leaf nodes and the internal nodes, giving the data/indexes of internal nodes, a value which the leaf nodes will never take (for us it is -1,). The links for a internal nodes is given by a dictionary. Hence, we get the following structure for a node in the Suffix Tree.

```
class Node():
    def __init__(self,title=-1,word=-1):
        self.out={}
        self.title=title
        self.word=word
```

Now that the structure of each node has been defined, we can go ahead with the creation of the suffix tree.

B. *Adding words*

There are two cases to take care of when we are adding words to a suffix tree i.e. when the word already exists in the suffix tree and when the word isn't already in the suffix tree.

1. *Word not in Tree*

When the word is not in tree, a new entry has to be created in the root node's dictionary and a new node has to be assigned for that word with the indexes. In the dictionary of the root node, a '$' is also added to the word as the key for the dictionary to signify that it is a leaf node.

2. *Word in Tree*

When the word is already present in the tree, it has two more cases that it can go through.

a) *Common prefix with another word*

When the word occurs as a substring of another, the node at which it arrives after coming down from the root will have to be replaced with an internal node, which branches out to two other nodes. One of them being for the new node for the word being added and the other one for the word already existing on the suffix tree.

b) *Word exists already as a node*

When the word already exists as a node in the suffix tree, then a list is created for the same key at the node which holds all the leaf nodes for a particular word.

C. *Adding suffixes of a word*

Adding suffixes of the word are done the same way that the words are added to suffix tree. The **add_word()** method does this automatically by getting all the suffixes of the word and adding them to the suffix tree.

To make it easier to take care of the results, while initializing the tree, the data is passed to it, taking care of all the addition of words in the data to the suffix tree. At the end of all additions, it reports how many words have been added to the suffix tree.

Notes:-

- **Words and not sentences are added to the suffix tree as they would give a lower height to the suffix tree.**
- **It is recommended to pass the data after running the get_data() and verify_data() so that no undefined behavior is observed.**
- **It is also recommended not to use the add_word() method directly without looking up the __init__ of the GStree class. Technically, it will still add the word to the tree, but it will give undefined behavior when check_word() is used along with it.**

V. SEARCHING FOR A WORD

Searching through the suffix tree is simple. Starting from root node, we take the entire word first, checking if it has a prefix matching with any of the keys of the root node, if it doesn't then the word doesn't have any occurrences. If the prefix matches with one of the keys, it goes down that node. When it goes down that node, the word to be checked for now becomes the word without the common prefix. This keeps going on till the word to be checked becomes an empty string. When it becomes an empty string, it needs to check in the current level and all levels below the current level for endings. Now let's look at how we implement this searching for a word in the suffix tree.
The **search()** method takes a word, checks if more than one word is being passed to it and then checks if the word is present using the **check_word()** method and then prints out the occurrences of the string in the document.
The **check_word()** method brings it to the node at which the word to be checked becomes an empty string. To get all the occurrences below that node and the node at the current level, a special method **get_all()** is used which has another method **get_all_inner()** which recursively calls itself to get all the leaf nodes at the current level and all the levels below the current level.
The **check_word()** method puts all the occurrences into a list, the occurrences being represented in a tuple where the first value of the tuple shows which story it belongs to and the second value of the tuple represents the word number in the story that it maps to.

The search() method then gets a 10 word locality of the occurrence of the matched word in the story and prints each occurrence in this way.

Notes:-

- **The check_word() method can be called directly but it will return the list of occurrences (tuples of each occurrence) and not the occurrences by printing them out where they occur in the document.**
- **The search() method doesn't currently support searching for more than one word. This may a feature implemented later or not at all.**

VI. First occurrence or longest substring

Finding the first occurrence of a given word can be done by first calling the **check_word()** method and taking the results and putting them into a dictionary where the story numbers are the keys and the values are a list of word numbers of the occurrences. Either getting the minimum of this list using the inbuilt **min()** method or sorting the list using the inbuilt **sort()** method and getting the first element from it is another way to get the first occurrence of a word if it occurs in the story.
This is done in the method **first_occur()**, which calls **check_word()** and does the above actions. When an occurrence is not found for a particular story, it calls **diff_check()** method. **diff_check()** method first finds all the substrings of the given word. It then checks for occurrences of substrings of the same length (i.e. "olf" or "wol") by calling the **check_word()** method and getting their occurrences. Once we get the occurrences, it checks if any of the occurrences occur in the particular story. A point to note here is that it only checks for all substrings of the same length at a time and not all the substrings at once. If it finds a suitable occurrence (i.e. belongs to the particular story) it returns a sorted list of occurrences, without trying to find occurrences of substrings smaller than the current length.
After this is done, it prints out the first occurrence or the first occurrence of the longest substring of the word given to **first_occur()** method.
Since it may also be possible that no substrings of a word might occur in a story, if a story doesn't have any occurrences, no occurrences will be printed out for it.

Notes:-

- **When trying to get the longest substring, generating all the substrings may not be necessary but it is done for simpler implementation.**
- **Only searching for prefixes or only searching for suffixes may seem like a good idea to start with, but not finding any occurrences on the first go might lead to missing out on some substrings completely.**
- **One can also use a modified check_word() method to get longest substring of a given word but it involves going up the tree which is not recommended.**
- **The diff_check() function can also be used separately to get longest substring of a word (not an occurrence of the word itself) in a particular story.**
- **The method first_occur() also doesn't support more than word being passed to it, so passing more than a single word will lead to undefined behavior. (will direct you to use rel() instead.)**

VII. Occurrences by relevancy

Before finding the occurrences of a given sentence by relevancy, we first need to define relevancy.
To make a simpler implementation, we will be using a simple relevancy definition. There are two criteria for being the most relevant occurrence in our document:-

A. Number of Occurrences

This is a very common criterion used by most algorithms but this can be misleading. If a single word occurs too many times it may overshadow the most relevant result. Care has to be taken to not assign a very high importance to this criterion. Nevertheless, we will still be using this as it is easier to implement and also gives a slight weightage to the occurrences of words.

B. Minimums and Maximums

This is an experimental method and may or may not have been tested before but we will still work with it. This is a bit more complicated than the number of occurrences but may add a bit more relevancy compared to the number of occurrences.
Here, we will compare the first occurrence of a word in the sentence with the first occurrence of a word before it, seeing if they are in some sort of order. We will also be comparing the last occurrence of a word with the last occurrence of a word before it. In short, we can say we are comparing minimums and maximums of two adjacent words.
While comparing the minimums, we must make sure that if the minimums of two adjacent words in a sentence are closer together, they are more relevant but also making sure that the minimum of the word before doesn't occur after the minimum of the word currently.
Similarly, when we compare maximums, we must make sure that if two maximums are closer together, they are more relevant while also making sure that the maximum of the word before isn't more than the maximum of the current word.
Theoretically this can also be used to check across for all words before a current word but we will be using this method for adjacent words in the sentence only.

Now that we have defined what is a relevant result in our document we can go ahead and look at the implementation of the occurrences by relevancy.
We will use the **rel()** method to do the occurrences by relevancy and show the occurrences by the definition of relevancy described above.
First the **rel()** method checks if the sentence being passed to the **rel()** method has more than one word or not. If it has only one word, it calls the **search()** method instead. If not, it goes ahead and tries to get occurrences of each word in the

sentence. Once this is done, it removes stories from the dictionary in which none of the words given in the sentences occur even once.
Now that we have a set of stories in which the words in the sentence occur, we can go ahead and assign scores to different stories to show how relevant they are. This is done by the **get_scores()** method which gives a score to each story depending on the criteria described above and returns a dictionary in which the scores are keys and the story numbers are the values or a list of story numbers are the values.
The **rel()** method then prints out the title of the stories and the occurrence of the word in them (i.e. the sentence or a word). A point to note here is that, the more relevant an occurrence is, the higher is the score it gets from **the get_score()** method.
It is also possible that not even a single word in the sentence is present in any of the stories. In this case, the **rel()** method prints out "No occurrences found."

Notes:-

- **When we have a single word being passed to the rel() method, we don't use any of the criteria as just finding out all the occurrences of the word using the search() method is a good enough way to find the most relevant occurrences. It is also possible to implement the most number of occurrences as the only criteria for relevancy for a single word to be searched but we have not implemented it and this could be a future feature of this.**
- **It is not recommended to use to get_scores() method directly to get the scores of a sentence as some preprocessing is done by the rel() method to pass on some useful data to the get_scores() method. If not used properly, the get_scores() method can give undefined behavior and may cause some runtime errors in your program.**
- **If you look at the get_scores() method, while assigning scores to different stories based on the minimums and maximums criteria we use the length of the stories (number of words in the story) and then subtract the difference of the minimums or maximums from it. While this may seem weird, it brings in a new criterion for relevancy. It says if the words are closer in larger stories, they are more relevant than if they are closer together in smaller stories.**

VIII. Conclusion

We conclude by looking at the time complexity of each of the methods described in the above sections.
The **add_word()** method has a time complexity of **$O(n^2)$** (assuming that getting the common prefix takes constant time, where n is the length of the word being added.
The **check_word()** method has a time complexity of **$O(n + m)$** where n is the length of the word and m is the total number of occurrences of the word. (same assumption as above)
The **first_occur()** method has a time complexity of **$O(n^2)$** where n is the length of the word. (finding the substrings)
The **rel_method()** has a time complexity of **$O(k*(n + m))$** where k is the number of words in the sentence, n is the length of each word and m is the total number of occurrences of each word.

IX. Availability

All the methods for the suffix tree are implemented in the **classes.py** file and the data clean methods are in the **data_clean.py** file. An example implementation of usage of the suffix tree is present in the **client.py** file which acts as a menu driven program.
These files will either be bundled with this report or can be found here. The input file AesopTales.txt is also included as part of this bundle.

X. Acknowledgements

I would like to thank Asst. Prof. Channa Bankapur for providing the input file "**AesopTales.txt**" with the formatting described in this report.

XI. Bibliography

[1] Wikipedia
[2] Google
[3] Geeks For Geeks

Arun John is a student currently doing his Bachelors in Computer Science and Engineering, People's Education Society University in Bangalore.