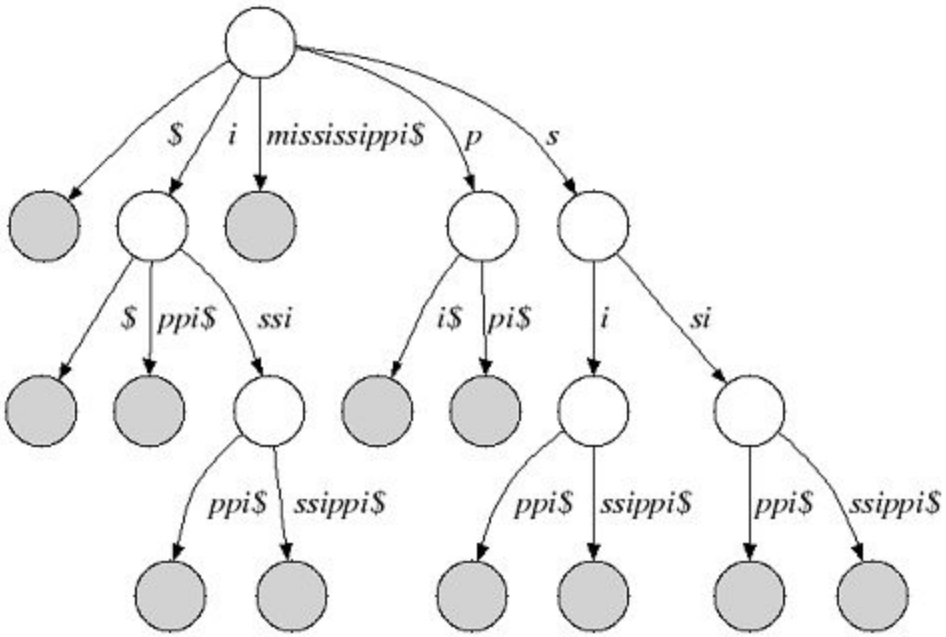
Advance Algorithms (Prof. NSK)

SUFFIX TREES

Shahid Ikram -- 01FB15ECS274



A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.

Here we build a Suffix tree for the given dataset which is a document of tales.

PRE-PROCESSING THE DATASET

I have preprocessed the dataset document to generate a dictionary in python where,

key: The tale-title value: The tale

This is done using the code in *Data_Processing.py*. After creating the dictionary, I have pickled the dictionary for further use so that there is no need to process and create the dictionary every time we want to build the tree.

BUILDING THE TREE

I have used the naive method for building the tree.

Naive method:
1) Generate all suffixes(terminated with '$') of given text.

2) Consider all suffixes as individual words and build tree for that suffix.

For each suffix, start at the root node and match the longest prefix of the suffix, moving down the tree and chop off that part of the suffix.

if leaf node is reached:

Add that suffix to the leaf

Else

Split the node that is reached and add a new edge,

Followed by a leaf node for that new suffix.

Complexity of this algorithm

At step i the algorithm finds the longest common prefix between S[i..s] and the previous i suffixes S[1..s]$, . . . , S[i-1..s]$. Let S[i..k]$ be such prefix. Then k+1 comparisons have been performed to identify such prefix. After that, the remaining of the suffix, S[k + 1..s]$ has to be read and assigned to the new edge. Therefore, the total amount of work at step i is θ(i).

Hence, the overall complexity is $\sum_{i=1}^{s} \theta(i) = \theta(s^2)$.

Definition of a node

```
class node:
        def __init__(self):
                self.suffix_num = {}   #tale_title(key), suffix_number(value)--Relevant only in leaf
                self.branches = [{}] * 256
        def set_suffix_num(self, tale_title, num)
        def add_branch(self, sub_string, child)
        def all_leaves(self)            # returns all the leaves of the node its called on
```

BUILDING A GENERALIZED SUFFIX TREE

The 1st part of the task requires to return all the occurrences of a pattern that matches across all the tales.

This requires the creation of a generalized suffix tree for all tales.

HOW?

Create a tree for a tale and then to obtain a generalized suffix tree, build the tree for the subsequent tales on the same root.

This creates a generalized suffix tree for all the tales.

I've not included the tale titles while building the suffix tree. Generalized suffix tree is built for the tales only.

PATTERN MATCH

Once the tree is built, matching a pattern is easy and fast.

The pattern matching problem can be solved in optimal O(m+k) time using the Suffix Tree, where k is the number of occurrences of Pattern in Text.

Suppose Pattern occurs in Text starting from position i. Then, Pattern is a prefix of suffix in Text. It follows that Pattern matches the path from root to leaf labelled i in the suffix tree.

This property results in the following simple **algorithm**:

Start from the root and follow the path matching characters in Pattern, until Pattern is completely matched or a mismatch occurs. If Pattern is not fully matched, it does not occur in Text. Otherwise, each leaf in the subtree below the matching position gives an occurrence of Pattern.

As the number of leaves in the subtree is k, this takes O(k) time. The problem of whether Pattern occurs in Text or the problem of finding one occurrence can be answered in O(m) time, where m is length of pattern.

Here, I have considered **strict case sensitive pattern matching** scheme i.e., Pattern Wolf is different from wolf

THE HEURISTIC USED FOR ASSIGNING RELEVANCE TO TALES

For a given query string, I have defined relevance for a tale based on two criteria:

1. For a query string of words I count the number of words of the query that are present in a tale irrespective of punctuations.

a. Higher the no. of matched words higher its (tale's) relevance
b. Now that we know how many words from the query has matched the tale text, we can group the tales based on the no. of matched words. Now within each group I use another factor to rank the tales.

2. Another criteria will now check, out of the no. of words from the query that matched with that tale, what is order of the matched words. It considers the maximum number of words in order.
   a. For the tales in which certain equal no. of words of the query exist, The relevance is resolved using this heuristic. The tale in which more number of words are in order is given higher relevance than the other.
   b. Punctuations are given importance here.
      i. For Example:
         1. Text = Hello, **I am from Bengaluru**
         2. Pattern = Hello **I am from Bengaluru**

         Here, Only the colored words are in order since Hello, isnt same as Hello (Because of comma).

         But the punctuation at the end of text is ignored

         Text = Hello, **What is your name**?

         Pattern = Hello **What is your name**

         Colored part is considered to be in order.