REPORT

Ganesh K. (01FB15ECS104)

Abstract:

To implement a Suffix tree to search for occurrences of a string in a document. The suffix tree is implemented in C++ using the Ukkonen's algorithm.

Objectives

The following problems to be solved for given dataset *AesopTales.txt*:

- List all the occurrences of a query-string in the set of documents.
- 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.
- For a given query-string (query of words), list the documents ranked by the relevance.

Approach:

The following steps were followed to build the tree:

Source: Geeks for Geeks:

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
    For j from 1 to i+1
        begin {extension j}
        Find the end of the path from the root labelled S[j..i] in the current tree.
        Extend that path by adding character S[i+l] if it is not there already
    end;
end;
```

Suffix extension is all about adding the next character into the suffix tree built so far. In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 extension rules:

Rule 1:

If the path from the root labelled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2:

If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1]. A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3:

If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

Document Ranking Heuristic:

A score is given to each document based on the following criteria with an initial score of 0:

- If all m characters of the query match then score is given as $(m \times 2)$.
- If a word matches (whole) k times from the query and n such words match, score if given by $\sum_{i=1}^{n} k_i \times x$.
- x is calculated as:
  - X = 2 if word is not a stop word.
  - X = 0.5 if words is a stop word.
- Thus, total score = $\sum_{i=1}^{n} (k_i \times x) + (m \times 2)$.

[IMPORTANT] Reasons for this ranking is as follows:

- If the user is searching for a phrase and a document has it as a whole, it must be given a very high priority. Hence the +m*2.
- If the user is searching for a query but is unsure of the sentence, he can search for the words present in the document he needs.
- Since the program is unsure if the user is going for case 1 or 2, the 'X' factor takes care of misinterpreting case 1 as case 2 by giving low score to matched stop words.

Example:

User searches: "*cats in the cradle and the silver spoon*":

- The user is going for option one and is searching for a sentence match.
- If such a sentence exists, the document will get a +39 which is significant.
- But if the sentence does not exist, and a document has cats, cradle and spoon, it will get a +6.

User searches for *“cat kitten dog puppies”:*

- The user is going for option two.
- Since there are no stop words, the document with the most matches to these household pets will get the highest score.

Git Link: https://github.com/DarkFate13/suffix-tree

Average Timing: (query: “*cat kitten dog puppies”,* Done 15 times on win-bash shell)

Problem	Average Time with print (ms)	
List all the occurrences	Build: 1239.8	Find: 0.0224
List only the first occurrence	1821.32	
List document relevance	1074.89	

Complexity:

Problem	Space	Time
List all the occurrences	O(N)	$O(N+M+Z_1)$
List only the first occurrence	O(M+N)	$O(M+N+Z_1+Z_2)$
List document relevance	O(N)	$O(N+M+(m_i+K+N))$

Explanation:

1. *List all occurrences:* N is size of entire document, M is size of query as we need to go through the length of the pattern. Z_1 refers to the number of suffixes we have to traverse or in terms of the tree, it’s the complexity of DFS of Z_2 leaves.

2. *List only the first occurrence:* N is size of each document, M is size of query. Since a GST is made, size shall be N+M. Hence DFS on such a tree to all matched leaves will be $M+N+Z_1$. Z_2 refers to the complexity of the min() function as we need to obtain the first occurrence.
3. *List documents by relevance:* Same as case 1 in terms space. We need an extra (m_i+k) as m_i refers to each word in query which occurs N times

Acknowledgement:

- This code is based on Geeks For Geeks implementation of [Suffix tree] using C.
- I would like to thank my professors, Prof. N.S. Kumar and Prof. Channa Bankapur for their valuable advice.
- I would like to thank [Tushar Roy] for his [video] on Ukkonens implementation of Suffix Tree.