Assignment 2 **Rutha Prasad**

UE15CS311 **01FB15ECS367**

Question Statement:

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.

Approach/Heuristics:

- A global array of pointers-to-strings is created to store all the tales. The input file is scanned line by line, where a new tale is detected when the first 2 words/atleast first 3 characters of the line are ALL CAPITALS (This occurs at the start of each tale). The previous scanned line is stored as the title and the rest of the tale is scanned till the next tale occurs.

- Suffix Tree implementation is a combination of **SUFFIX ARRAY** implementation and a **Lowest Common Substring (LCP) supplementary array** which helps in searching.

- A global array of pointers-to-arrays is made to store all the Suffix arrays. The tales are scanned for suffixes, and each suffix starts with a character.(θ (n)) The array of **suffixes are sorted** and stored in the array such that they are in **lexical order**.(θ (nlogn))

Thus building Suffix Arrays:

Space complexity: O(number of tales*n approximately characters)→ θ (n)

Time complexity: O(number of tale*store n approximately characters+sorting)→ θ (n+nlogn)

- During the search-algorithms , a **binary search function** is used on the selected tale's suffix array to find the first occurrence of a possible match of the pattern in the tale.(O(logn)). And the **range(r) of indexes** surrounding the first occurrence found is calculated(O(r)). Then the r suffixes are searched for a match. Additionally for substring searching, **LCPs are dynamically created for each tale** with the pattern and used to find the longest common substring.

Thus searching Suffix Arrays:

Space complexity: O(pre-built suffix array+dynamic LCP array of length of pattern)→O(n+k)→O(n).

Time complexity: O(binary search+ranging+substring ranging)→θ(logn+r+m^2)→θ (logn+m^2)→O(n*m).

- The **rank of each tale is incremented for each occurrence**, based on the following categories out of the given guidelines:

  (a) Sequence of text matches pattern is more relevant than subsequence of text that is, all the words in query appear in the same relative order.

  (e) Whole string matched is more relevant than a subset of the string found

  (f) Exact word matched is more relevant than a similar word found.

Algorithms and Complexity:

Algo ALL_OCCURENCES(all_tales[T_1…..T_t], pattern[0……m])

```
{       For each tale $T_i$ in all_tale [$T_1$….$T_t$] ……….. O(t)
        {
                Index ← Binary search return_index [$T_i$[0…..n],pattern[0……m])……….. O(logn)
                Range[0…r] ← left and right indexes of Index……….. O(r)
                For each index in Range……….. O(r)
                        Found←true /false if m characters match. ……….. O(m)
                Return Title ……….. O(1)
                Return Context of occurrence. ……….. O(1)
                Increment and Return Rank of tale. ……….. O(1)
        }
}
```

Time complexity: θ(logn+ r+ (r*m)+ 1+ 1+ 1)→θ(logn+r+r*m+3)→O(logn+r*m)

Worst case= O(n) **Best case=O(logn)** **Average Case=O(logn+m)**

Assignment 2 **Rutha Prasad**

UE15CS311 **01FB15ECS367**

<u>Algo FIRST_OCCURENCES(all_tales[T_1.....T_t], pattern[0……m])</u>

```
{       For each tale $T_i$ in all_tale [$T_1$….$T_t$] ……….. $O(t)$
        {       (Create LCP)
                For each character in pattern[0……m] ……….. $O(m)$
                {
                        Index ← Binary search return_index [$T_i$[0…..n],pattern[0……m])……….. $O(logn)$
                        Range[0…r] ← left and right indexes of Index……….. $O(r)$
                        For each index in Range……….. $O(r)$
                            $LCP_i$[0….m]←append longest common prefix length after matching…….. $O(m)$
                }
                Time complexity: $O(m(logn+r+r*m))=O(mlogn+m*r+m*r*m))=O(mlogn+m^2)$
                (Search LCP)
                For each index in $LCP_i$[0….m] UNTIL first occurrence is not found ……….. $O(m)$
                {       Found←true /false if m characters match. ……….. $O(m)$
                        Return Title ……….. $O(1)$
                        Return Context of occurrence. ……….. $O(1)$
                        Increment and Return Rank of tale. ……….. $O(1)$
                }
                Time complexity: $O(m*m+1+1+1)=O(m^2)$
}
```

Time complexity: θ (mlogn+m^2 +m^2)→ O(mlogn+m^2)

Worst case= O(n*m) **Best case=O(m)** **Average Case=O(mlogn+(m^2/2))**

Algo RANK(all_tales[T_1.....T_t], pattern[0……m])

```
{       As ranks of each tale are decided by above two algorithms.
        If all_occurences_called=false
                all_occurences( all_tales[$T_1$.....$T_t$], pattern[0……m])………………*O(logn+m)*
        If first_occurences_called=false
                first_occurences( all_tales[$T_1$.....$T_t$], pattern[0……m]) ………… *O(mlogn+$m^2$)*
        Rank[$T_1$.....$T_t$]←sort(Rank[$T_1$.....$T_t$]) ………………*O(tlogt)*
        Display (Rank[$T_1$.....$T_t$] ) ………………*O(t)*
}
```

Time complexity: θ(logn+m+mlogn+m^2+tlogt+t)→ θ (mlogn+m^2+tlogt)→O(n*m+ tlogt)

Conclusion:

If the complete pattern has to be found, it is found in not more than linear time.

If substrings have to be found, they are found in not more than quadratic time.

Comparing to other string matching algorithms

Brute Force < Horspool < Boyer Moore == Suffix Tree == KMP == Rabin Karp