Trie and Suffix Array Datastructures

German University in Cairo

March 11, 2013

Outline

Outline

Why to learn more after Z-function??

- Finding a word in a dictionary efficiently.
- Longest common substring problem (computational biology, plagiarism detection.. etc.).
- Longest K-repeated substring.
- and many more actually ...

Outline

Historical note

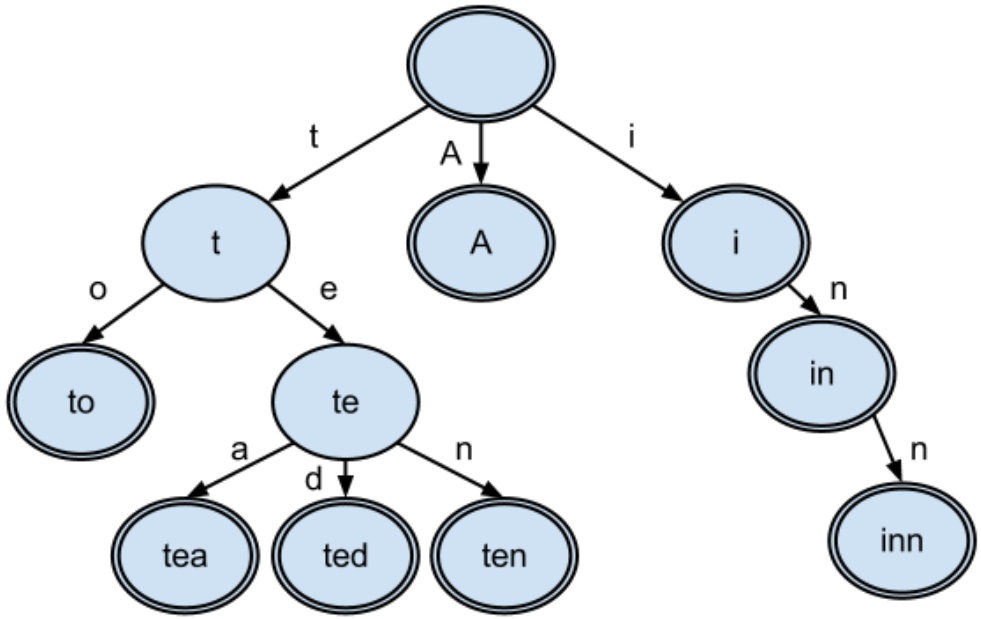
- The name ”Trie” came from the use of the datastructure for re*trie*val. People used to pronounce it like ”Tree” but recently the dominant pronounciation is like the word ”Try”.

Semi-formal Definition

- A Trie is a tree for storing strings where there is only one node representing a common prefix and every edge represents a symbol in the considred alphabet. Traversing the tree starting from the root till a node while concatenating the symbols on the traversed edges yields the common prefix for that node.
- From that we can conclude that a leaf node should correspond to a whole input string.

Trie example

- Here is an example of a Trie after inserting the words: ”A”, ”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, ”inn”. Nodes represented with double circles denote full words.



Searching a Trie

- Searching a Trie is straight forward. Start from the root while reading characters from the string reuqiured to be found.
- According to the string's current character, choose the corresponding child of the current Trie node or report that the string doesn't exist (if the required child doesn't exist).
- When you reach the end of the string:
  - If the current node is labeled as a word node then the string was found.
  - Else the string is not there.

Outline

Semi-formal definition

- A suffix array of string is the lexicographically sorted array of the string suffixes. For example if the string is ”bobocel” then the sorted array should be:
  1. bobocel
  2. bocel
  3. cel
  4. el
  5. l
  6. obocel
  7. ocel

Construction Algorithms

- Construction algorithm for suffix array vary widely in its worst case running time from $O(n^2 \lg n)$ down to $O(n)$.
- Although linear time algorithm exists, the constant factor is just so big that a theoritically less efficient algorithms perform better in practice.
- Algorithms with running time $O(n \log^2 n)$ and $O(n \lg n)$ are pretty sufficient for most application and more importantly very easy to code compared to linear time algorithms.
- We will consider the $O(n \log^2 n)$ algorithm, which can be easily tweaked to be $O(n \log n)$.

Algorithm

- let's denote the substrting of length 2^k which starts at position i by S_i^k i.e. it is the substring from position i to $i + 2^k - 1$.
  Note that the index $i + 2^k - 1$ may exceed the string length but we'll assume that the string is padded by arbitraty characters that are lexicographically smaller than all characters in the string.
- The algorithm starts by first sorting the characters of the string where each character represents the suffix starting at its position. So we can say that after this step the suffixes are sorted according to their first character.
- The algorithm then performs $\lceil log\, n \rceil$ iterations where n is the length of the string.

Algorithm

- The loop invariant is that at the start of each iteration the suffixes become sorted according to their first 2^k symbols.
- It is clear that if the loop invariant is properly maintained then at the end the suffixes will be sorted according to their first $2^{\lceil log\, n \rceil}$ symbols which is at least n , i.e. the required sufffix array is formed.

Maintatining the loop invariant

- Achieving the loop invariant at iteration $k+1$ will use the fact the suffixes are already sorted according to the first k symbols from iteration k.
- Three cases may appear when comparing two suffixes (one starting at i and another starting at j) according to the first $2k$ symbols:
  - S_i^k is lexic. less than S_j^k then S_i^{k+1} is lexic. less than S_j^{k+1}.
  - S_i^k is lexic. greater than S_j^k then S_i^{k+1} is lexic. greater than S_j^{k+1}.
  - S_i^k lexic. equals to S_j^k so their order will be determined based on the next k symbols i.e (from $i+k$ till $i+2k-1$ and from $j+k$ till $j+2k-1$) but we already know this relation from the previous iteration , namely we know the relation between S_{i+k}^k and S_{j+k}^k so this will be the same relation between S_i^{2k} and S_j^{2k}.

Maintatining the loop invariant

- Simple mathematical induction can show that the loop invariant can be easily maintained like that till the end of the loop.

Implementation

- One Possible way to implement what we discussed so far is having a matrix which stores for every suffix its position at every iteration.
- the sorting at iteration k will check for the three cases we have seen and will compare the suffixes accordingly.
- Notice that when two suffixes are found to be equal they will be given the same position in the partailly sorted array.
- Since the we have exactly $2^{\lceil log\ n \rceil}$ iterations and at every iteration we make one call to a sorting algorithm, then the time complexity is $O(n\ log^2\ n)$.

Implementation Tweaks

- The described algorithm can be easily modified for a run time complexity of $O(n \log n)$. How??
- The space complexity of the algorithm in $O(n \log n)$ but it can also be reduced to $O(n)$. How??

Outline

References

- Stanford Course: Introduction to competetive programming contests