Implementation and Analysis of EERtree

November 7, 2022

Kartik Tiwari (2021CSB1102) ,
Sahil (2021CSB1128) ,
Prashant Singh (2021CSB1124)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sravanthi Chede

Summary: This project implements an EERTREE, an online, linear-size graph based data structure which provides quick access to all the palindromic substrings of a string. This structure inherits some ideas from the construction of both the suffix trie and suffix tree. Using this data structure, we then present efficient solutions of a number of palindrome related problems such as insertion into the tree, deletion of characters, printing distinct palindromes and their number of occurences in the strings and number of palindromes ending or starting from a given index.

1. Introduction

Palindromes are important repetitive string structures. They have been actively studied in combinatorics, language theory and applied in practical life in fields like bioinformatics to analyse DNA and nucleic acid structures. Any string $S = a_1a_2..a_n$ which is exactly equal to its reversal $\overleftarrow{S} = a_n..a_2a_1$ is known as a palindrome.

The counting of distinct palindromes in general and from certain indices in a string has been a problem of much interest since a long time. The data structure presented here, published in 2018 (see [2]) provides a new tree like data structure which simplifies and speeds up several palindrome related problems. In the following sections, we will look upon the functioning of this tree.

In the following subsections, we look upon the terminology used for the data structure and the functions implemented for it.

1.1. Definition and notation

The data structure deals with finite strings, treating them as arrays of symbols : S = S[0..n]. The notation σ stands for the number of distinct symbols in the processed string. For the sake of implementation, we use the 26 alphabets of the English language, i.e, we restrict σ to 26. ϵ represents an empty string, and |S| represents the length of the string S.
S[i] represents the i^{th} character of the string and S[i,j] represents the substring s[i]s[i+1]...s[j]. By definition, s[i,i-1] = ϵ for any i. ϵ is not considered as a palindrome.

1.2. Functions implemented

The functions implemented here using this data structure are :
1. Insert() - appends a given string into the eertree
2. Multipop() - removes characters from the string and accordingly, nodes from the eertree

3. Print() - prints all the distinct palindromes and their respective occurrences
4. StartAtIndex() - prints number of distinct palindromes starting at that index
5. EndAtIndex() - prints number of distinct palindromes ending at that index

2. Algorithms

2.1. Insert

The **insert(c)** function appends a character c to the processed string and updates the tree accordingly, and returns the number of new palindromes that appeared in the string. This value is always 0 or 1 *(See Lemma 1)*.

An eertree is a directed graph with some modifications to its nodes. Each node stores the length of palindrome it refers to, along with start and end index of the palindromic substring in the original string. For initialization purposes, two special nodes are added: with the number 0 and length 0 for the empty string, and with the number -1 and length -1 for the "imaginary string".

The edges of the graph are defined as follows. If c is a symbol, v and cvc are two nodes, then an edge labelled by c goes from v to cvc. The edge labelled by c goes from the node 0 (resp. -1) to the node labelled by cc (resp., by c) if it exists.
An unlabelled suffix link link[u] connects u to v if v is the longest proper suffix-palindrome of u. By definition, link[c] = 0, link[0] = link[-1] = -1.

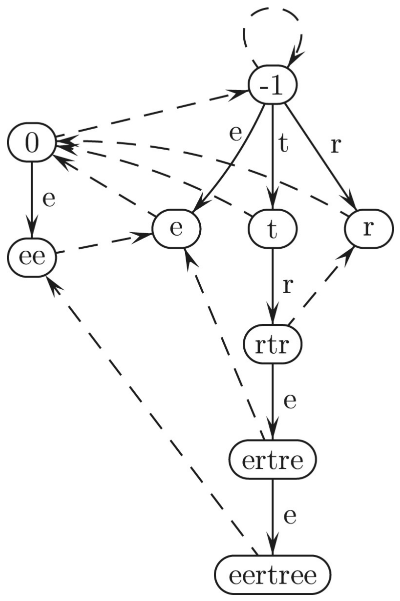


Figure 1: The eertree of the string *eertree*. Edges are solid, suffix links are dashed.

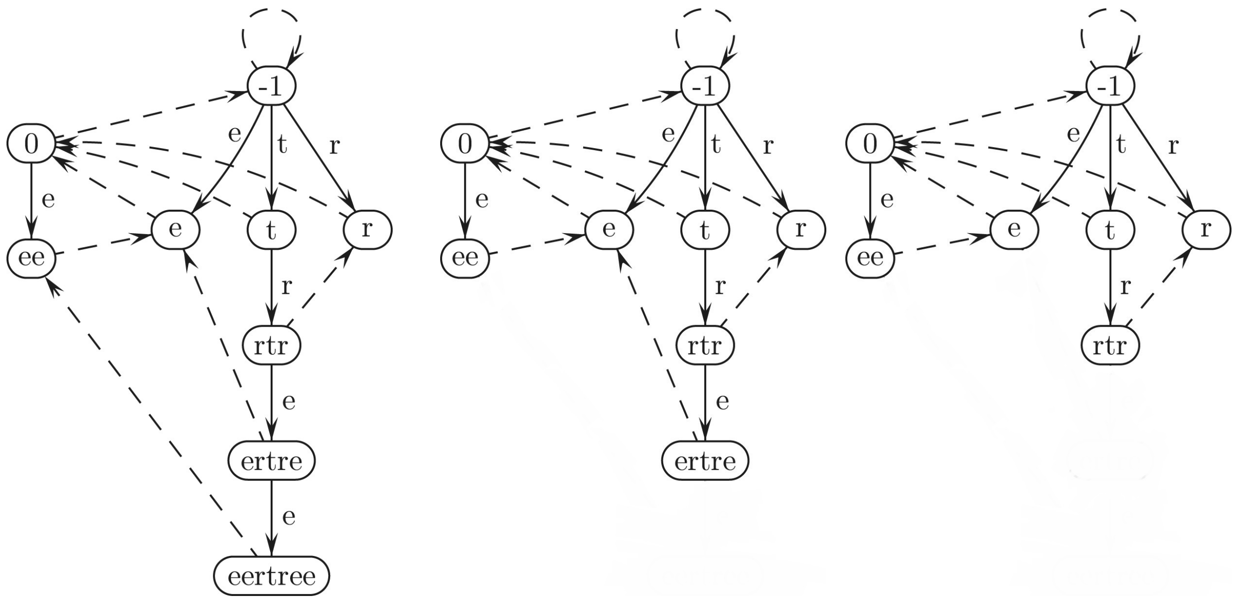
Lemma 1 Let S be a string and c be a symbol. The string Sc contains at most one palindrome, which is not a substring of S. This new palindrome is the longest suffix-palindrome of Sc. [3]

Lemma 2 A node of positive length in an eertree has only one incoming edge. Let S be a string and c be a symbol. The string Sc contains at most one palindrome, which is not a substring of S. This new palindrome is the longest suffix-palindrome of Sc.

Proposition 2.1. *For a string S of length n, eertree can be built online in O(n logσ) time with O(n) space [2]*

2.2. Multi Pop

To implement pop utility, we include an occurrence variable in each node which counts the number of times a palindromic substring has occurred in the given string.
The function checks if the given node is occurring more than once, and then after decreasing its occurrence, checks on the suffix links. In case of a single occurrence, the node is deleted after rearranging its suffix links accordingly.



(a) eertee for *eertree* (b) eertee for *eertre* (c) eertee for *eertr*

Figure 2: Using pop on last 2 characters

2.3. Print

Printing of distinct palindromes just traverses over the eertree recursively. The function is called from both root1 and root2, where root1 is the node with 0 length and root2 is the node with -1 length.

Algorithm 1 void print(string s, node* head)

```
 1: if $head! = root1$ or $head! = root2$ then
 2:     for $i = head \rightarrow start$ to $head \rightarrow end$ do
 3:         print s[i]
 4:     end for
 5: end if
 6: for $i = 0$ to $i = \sigma$ do
 7:     if $head \rightarrow label! = NULL$ then
 8:         print(s, head->label[i])
 9:     end if
10: end for
```

The function show() then takes this print function to iterate over the tree and print all unique palindromes. Since it just traverses the graph, the function operates linearly in O(n).

Algorithm 2 void show(string s)

```
1: print(s, root1)
2: print(s, root2)
```

2.4. Start and End at Index

We create a Prefix and Suffix array which is updated accordingly when the insertion procedure is applied, so for each index, we know the number of palindromes starting and ending at that index. From Lemma 1, we know that adding a character can only increment the number of palindromes by 1 at max. So the array is updated after checking if a palindrome is created.

3. Running time of the functions

Below is the runtime for input of various size for the various functions. The input is tested with three types of strings :
1. Type 1 - single character c repeated n times
2. Type 2 - palindromic string with more than 10 characters
3. Type 3 - Non palindromic string with different characters

	Type 1	Type 2	Type 3
Size: 100	0 ms	0 ms	0 ms
Size: 1000	0.3 ms	0.2 ms	0.1 ms
Size: 100000	29 ms	23 ms	18 ms

Table 1: Runtime for insert function

	Type 1	Type 2	Type 3
Size: 100	0 ms	0 ms	0 ms
Size: 1000	0.2 ms	0.1 ms	0.1 ms
Size: 100000	26 ms	20 ms	13 ms

Table 2: Runtime for pop function

4. Further suggestions

Proposition 4.1. *The palindromic length of a string can be found in O(n logσ) time online and in O(n) time offline. [2].*

Proposition 4.2. *The eertree of a length string over the alphabet can be built offline in O(n logn) time. [1]*

Proposition 4.3. *Using an eertree, the k-factorization problem for a length n string can be solved online in time O(n logn). [2].*

5. Conclusions

Implementation of the data structure eertree was done with insert, pop and print utilities, to name a few. The online data structure is prepared in O(n logσ) and takes linear space, with σ restricted to 26 here. The tree efficiently prints all distinct palindromes with their occurrences in linear time. Number of palindromes starting or ending at certain index can also be told in constant time. Append and pop is supported in this version of eertree.

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant Sravanthi Chede for their guidance and valuable inputs provided during the project.

References

[1] Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic Length in Linear Time. 78:23:1–23:12, 2017.

[2] Arseny M. Shur Mikhail Rubinchik. Eertree: An efficient data structure for processing palindromes in strings. *European Journal of Combinatorics*, pages 249–265, 2018.

[3] Giuseppe Pirillo Xavier Droubay, Jacques Justin. Episturmian words and some constructions of de luca and rauzy. *Theoretical Computer Science*, pages 539–553, 2001.