EER tree implementation and applications

November 7, 2022

Khushboo Gupta (2021CSB1105) ,
Nandini Mundhra (2021CSB1113) ,
Niroopma Verma (2021CSB1115)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Vinay Kumar

Summary: Palindromic tree or eertree is a linear-size data structure which provides a fast access to all palindromic substrings of a string or a set of strings. This structure inherits some ideas from the construction of both the suffix trie and suffix tree. Using this structure, we present simple and efficient solutions for a number of problems involving palindromes. It allows fast access to all Palindromes of a String.

---

1. Introduction

This data structure allows to keep track of palindrome substrings (also in an online fashion) in linear time and space (truth be told the time complexity is O(n*log(a)), where a is the size of the alphabet, but an alphabet usually contains a limited number of symbols, so we can consider log(a) a constant factor and hence the linear time O(n)).
The eertree for a given string is a directed graph with N+2 nodes, where N is the number of the distinct palindrome substring found in the initial string, and 2 are the degenerate palindromes that represents the roots of the graph . Our Motive problem is to find distinct subpalindromes online Well-known online linear-time Manacher's algorithm outputs maximal radiuses of palindromes in a string for all possible centers, thus encoding all subpalindromes of a string. Another interesting problem is to find and count all distinct subpalindromes. Groult et al.solved this problem offline in linear time and asked for an online solution. Such a solution in O(nlog) time and O(n) space was given in, based on Manacher's algorithm and Ukkonen's suffix tree algorithm. As was proved in, this solution is asymptotically optimal for a general ordered alphabet. But in spite of a good asymptotics, this algorithm is based on two rather "heavy" data structures. It is natural to try finding a lightweight structure for solving the analyzed problem with the same asymptotics. Such a data structure is described below. Its further analysis revealed that it is suitable for coping with many algorithmic problems involving palindromes.

2. EER TREE

In EER Tree for every node it is mandatory to store the length of the palindrome substring, but also additional informations such as the occurrences can be stored, for example if you need to know the total number of palindrome substrings including the duplicate ones.
Structure of Palindromic Tree
Palindromic Tree's actual structure is close to directed graph. It is actually a merged structure of two Trees which share some common nodes(see the figure below for better understanding). Tree nodes store palindromic substrings of given string by storing their indices. This tree consists of two types of edges :
1) Insertion edge (weighted edge)
2) Maximum Palindromic Suffix (un-weighted)
1. **Labeled edges directed downwards / Insertion edge (weighted edge)**: Basically , this is a directed edge that faces downwards. It has some character as its label. A labeled edge with label a from node u to node

v means that node v can be obtained by adding a as a prefix and suffix to substring of u. This makes it possible to obtain a palindromic substring by traversing down from empty string node.

Figure 1: Insertion edge and suffix edge

Every node may have zero or more of these edges outgoing, and exactly one incoming, otherwise the palindrome would not be reachable.
2. **Unlabeled edges directed upwards / Maximum Palindromic Suffix (un-weighted)**: As the name itself indicates that for a node this edge will point to its Maximum Palindromic Suffix String node. We will not be considering the complete string itself as the Maximum Palindromic Suffix as this will make no sense(self loops). For simplicity purpose, we will call it as Suffix edge(by which we mean maximum suffix except the complete string). It is quite obvious that every node will have only 1 Suffix Edge as we will not store duplicate strings in the tree. We will use Blue dashed edges for its Pictorial representation.
Root Nodes and their convention:
This tree/graph data structure will contain 2 root dummy nodes. More, precisely consider it as roots of two separate trees, which are linked together.
Root-1 will be a dummy node which will represent a string of length -1, i.e. an imaginary string.if you put one character on both sides of it, its length will be 1, so it will be possible to store palindrome of odd length Following a labeled edge from this root adds only the single character that is in label. For example, following the edge with label a from this root will give the string 'a'. Root-1 has a suffix edge connected to itself(self-loop) as for any imaginary string of length -1 , its Maximum palindromic suffix will also be imaginary, so this is justified.

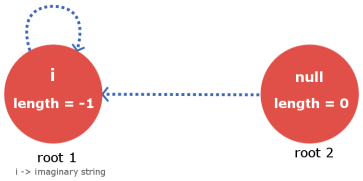
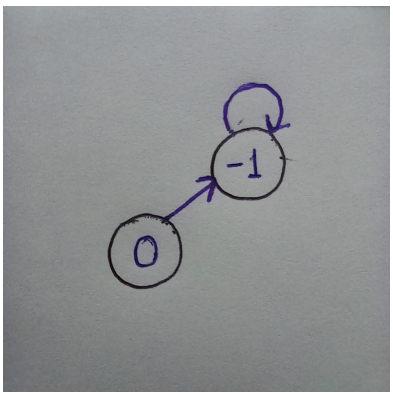


Figure 2: Root 1 and Root 2

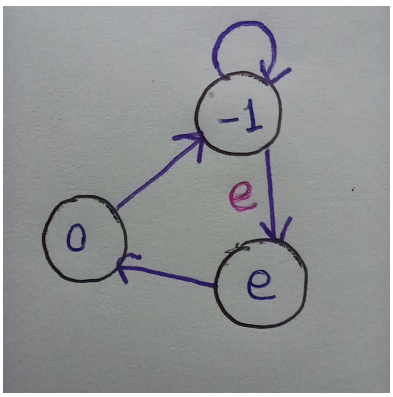
Root-2 will be a node which will describe a null string of length = 0.Second root is a string with length 0, i.e. an empty string,and if you add Character on both sides of it, even length Palindrome will be found. Labeled edge performs normally on this node. Now Root-2 will also have its suffix edge connected to Root-1 as for a null string (length 0) there is no real palindromic suffix string of length less than 0.

2.1. Building the EER Tree

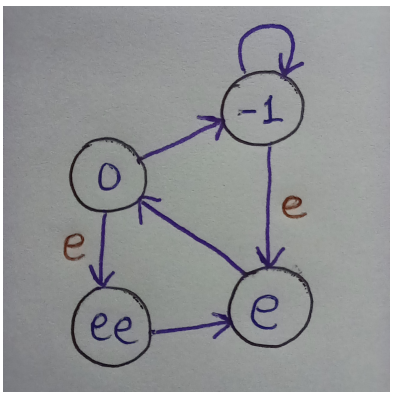
The step-by-step the execution of the algorithm in order to create an eertree that stores all the distinct palindrome substrings of the string "eertree" is as follows:
Step 1: Initialisation Initialize the eertree adding the two roots: one representing the empty string (length 0), and the other representing an imaginary string with length -1.
The "suffix links" (the unlabeled edges) for the two nodes are both pointing towards the imaginary node. This node will be used a stop point during the backwards traversing of the tree later in the algorithm's execution.
There are no labeled edges out from the two nodes yet, but we will add the edges while we scan through the string.
The reference to the current largest palindrome substring suffix (for convenience I will call this suffix cursor from now on) points to the empty string.



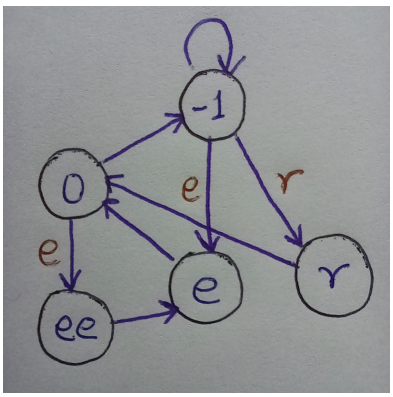
Step 2: Adding [e] from [e]ertree to the data structure We need to find the the next palindrome eXe to insert in the eertree: this is done through a process by that looks for the longest valid palindrome prefix X traversing the graph starting from the suffix cursor and moving upwards using the unlabeled nodes or "suffix links".
We start with X=[EmptyString], but e[EmptyString]e=ee is a palindrome but not the one we are looking for. We then use the suffix link from the [EmptyString] to the imaginary string: since the imaginary string it's theoretically centered at -1, we don't care about the "e" to prepend before the -1 position and we just care about the "e" appended at position 0, resulting in a new palindrome "e". We add then the new palindrome (a new node) "e" to the eertree, with a labeled edge "e" going from the longest prefix (the imaginary string node) to the new node.
We have now to set the suffix link for the new node: it represents the longest proper palindrome suffix eYe for the new palindrome "e" (proper means that it should be different from the whole palindrome).
Again, we obtain it by moving upwards on the eertree nodes through the suffix link: this time we start from the suffix link of the palindrome prefix used to build the new node: suffix([Immaginary String])=[Imaginary String]: in this case we apply a special rule (the base rule will be explained in the step 3) that is "whenever we should put the imaginary string as suffix link we replace it with the empty string instead".
We now update the reference to the largest inserted suffix palindrome to the new node "e" and we proceed to inserting the next letter in the eertree.



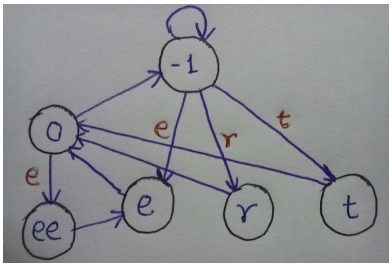
Step 3: Adding [e] from e[e]rtree to the data structure The largest palindrome prefix X such that eXe is palindrome is now the empty string, so we add the new node "ee" and a labeled edge "e" from the empty string to the new node.
The largest palindrome suffix Y such that eYe is palindrome is obtained traversing the graph backwards using the suffix links, starting from the suffix link of Y (the imaginary string): in this case prepending and appending "e" to the imaginary string we obtain a proper palindrome suffix for "ee", that is "e": this will be the suffix link for "ee".
We now update the reference to the largest inserted suffix palindrome to the new node "ee" and we proceed to inserting the next letter in the eertree.



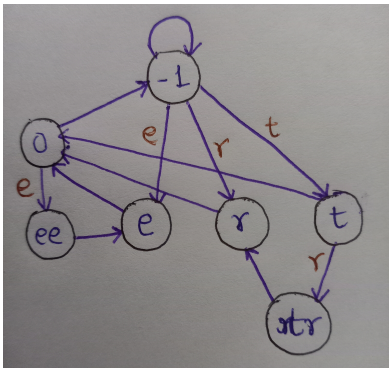
Step 4: Adding [r] from ee[r]tree to the data structure We start with X=”ee” (largest inserted suffix palindrome) and we check if rXr is a palindrome substring of “eer” (portion of the string examined so far). It is not, so we traverse the graph back: rXr is not a palindrome substring either for X=e (suffix link from X=ee), X=[Empty String] (suffix link from X=e), and we fall back to the special case with largest palindrome prefix equal to the imaginary string, already explained in step 2.
The new palindrome will be “r”, we find the suffix link in the same way it has been done for the “e” palindrome (that is applying the special rule and setting it to [EmptyString], then we update the reference to the largest inserted suffix palindrome to the new node “r” and we proceed to inserting the next letter in the eertree.



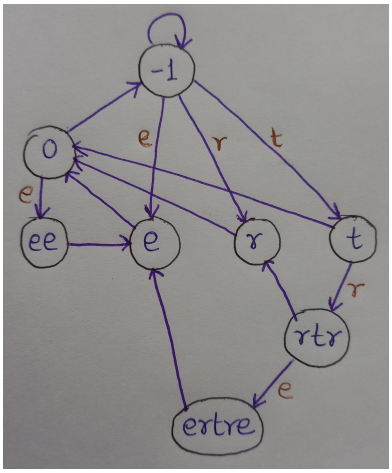
Step 5: Adding [t] from eer[t]ree to the data structure We are in the same situation as before, the largest palindrome prefix is the imaginary string, we add a new node “t”, and the largest palindrome suffix of “t” the empty string.
We update the reference to the largest inserted suffix palindrome to the new node “t” and we proceed to inserting the next letter in the eertree.



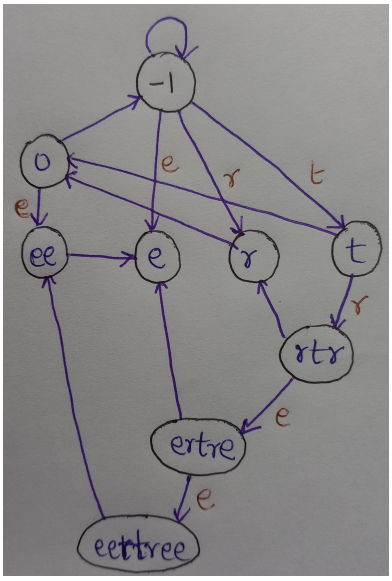
Step 6: Adding [r] from eert[r]ee to the data structure Finally things start to get interesting: for X=t rXr is a valid palindrome suffix, so can insert a new node representing the palindrome “rtr”, and from the largest palindrome prefix X=t we draw an edge to the new node “rtr” with label “r”.
We then look for the largest proper palindrome suffix rYr of “rtr”: we start with for Y=suffix link(X)=suffix link (t)=[Empty String]: r[EmptyString]r is not a valid palindrome suffix, so we traverse back the suffix link: set Y=suffix link([Empty String])=[Imaginary String]. In this case r[ImaginaryString]r is a proper palindrome substring: it corresponds to the palindrome “r”, so the suffix link for “rtr” will be an unlabeled edge going from “rtr” to “r”.
We now update the reference to the largest inserted suffix palindrome to the new node “rtr” and we proceed to inserting the next letter in the eertree.



Step 7: Adding [e] from eertr[e]e to the data structure For X=rtr eXe is a valid palindrome suffix, so we can insert in the eertree a new palindrome node "ertre". From the node representing the largest palindrome prefix X=rtr we draw a labeled edge "e" to the new node "ertre".
We then look for the largest proper palindrome suffix eYe of "ertre": we start with for Y=suffix link(X)=suffix link (rtr)=r : e[r]e is not a valid palindrome suffix, so we traverse back the suffix link (Y=suffix link(r)=suffix link(empty string): in this case e[empty string]e is a palindrome, but we can't get it because "ee" is not a suffix of "ertre": for Y=suffix link ([Empty String])=[Imaginary String] we get the palindrome suffix "e", drawing a suffix link from "ertre" to it.
We then update the reference to the largest inserted suffix palindrome to the new node "ertre".



Step 8: Adding [e] from eertre[e] to the data structure For X=ertre eXe is a valid palindrome suffix, so can insert a new node for the palindrome "eertree", and from the node representing the largest palindrome prefix X=ertre we draw a labeled edge "e" to the new node.
We then look for the largest proper palindrome suffix eYe of "eertree": we start with Y=suffix link(X)=suffix link (ertre)=e, but e[e]e is not a valid palindrome suffix, so we traverse back the suffix link: (Y=suffix link(e)=suffix link([Empty String]). Now e[Empty String]e is the palindrome "ee" and unlike in step 7 it is a suffix of our palindrome: we can draw a suffix link from "eertree" to it.



With this we have run out of new letters, so all the distinct palindromes has been added to the data structure.

2.2. Complexity

2.2.1 Time

Constructing a palindrome tree takes $O(n \log \sigma)O(n \log \sigma)$ time, where n is the length of the string and σ is the size of the alphabet. With n calls to add(x), each call takes $O(\log \sigma)O(\log \sigma)$ amortized time. This is a result of each call to add(x) increases the depth of the current vertex (the last palindrome in the tree) by at most one, and searching all possible character edges of a vertex takes $O(\log \sigma)O(\log \sigma)$ time. By assigning the cost of moving up and down the tree to each call to add(x), the cost of moving up the tree more than once is 'paid for' by an equal number of calls to add(x) when moving up the tree did not occur.

2.2.2 Space

A palindrome tree takes $O(n)$ space: At most $n + 2$ vertices to store the sub-palindromes and two roots, n edges, linking the vertices and $n + 2$ suffix edges.

2.2.3 Space–time tradeoff

If instead of storing only the add edges that exist for each palindrome an array of length σ edges is stored, finding the correct edge can be done in constant time reducing construction time to $O(n + p * \sigma)$ while increasing space to $O(p * \sigma)$, where p is the number of palindromes.

3. Algorithms and Applications

3.1. Manacher's Algorithm

Manacher (1975) invented an $O(n)$-time algorithm for listing all the palindromes that appear at the start of a given string of length n. However, as observed e.g., by Apostolico, Breslauer Galil (1995), the same algorithm can also be used to find all maximal palindromic substrings anywhere within the input string, again in $O(n)$ time. Therefore, it provides an $O(n)$-time solution to the longest palindromic substring problem. This algorithm is faster than the brute force approach, as it exploits the idea of a palindrome happening inside another palindrome.
Manacher's algorithm is designed to find the palindromic substrings with odd lengths only. To use it for even lengths also, we tweak the input string by inserting the character hashtag at the beginning and each alternate position after that .
In the case of an odd length palindrome, the middle character will be a character of the original string, surrounded by hashtag.
In the case of an even length palindrome, the middle character will be a hashtag character.

3.2. KMP algorithm

KMP algorithm is used to find a "Pattern" in a "Text". This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "Prefix Table" to skip characters comparison while matching. The KMP algorithm was the first-ever string matching algorithm that ran in linear time. Most of the naive string matching algorithms run in O(nm) time, while the KMP algorithm runs in O(m + n) time where n is the length of the string, and m is the length of the pattern.
Approach:
We make a pi Table here so keep track of common prefixes. We have to shift indices of the pattern by 1 to make the process easier, ie, the pattern begins at index 1 and we always compare pattern[j+1] with string[i] and not the jth index of the pattern. Then the only change is that whenever there is a mismatch, then we go to the index pointed to by our pi table and do not stop the search. Like this, we keep on going to previous indices of the pi table on mismatch until there is a mismatch or that j=0(refer to code for more understanding). So like this, when we find that j is m, we stop as we have found the pattern, else return -1 when i reaches n.
Pseudo Code for KMP Algorithm:
table[0] = 0
i = 0, j = 1
while (j < n) // n is the length of pattern p
if (p[i] == p[j])
table[j] = i + 1;
i++;
j++;
else
if (i != 0)
i = table[i - 1];
// Do not increment j here
else
table[j] = 0;
j++;

3.3. Number of distinct Palindromes

When the problem is to find and count all distinct subpalindromes , its solution can be given in O(nlog) time and O(n) space , based on Manacher's algorithm and Ukkonen's suffix tree algorithm . And, these solutions are asymptotically optimal in the comparison-based model. But in spite of a good asymptotics, this algorithm is based on two rather "heavy" data structures. It is natural to try finding a lightweight structure for solving the analyzed problem with the same asymptotics and that such data structure is EER Tree.
From inside, EERTree is a directed graph with some extra information. Its nodes, numbered with positive integers starting with 1, are in one-to-one correspondence with subpalindromes of the processed string. We write eertree(S) for the state of eertree after processing the string S letter by letter, left to right. So , to report the number of distinct subpalindromes of S, just return the maximum number of a node in eertree(S).

3.4. Longest palindromic substring using palindromic tree

The longest palindromic substring or longest symmetric factor problem is the problem of finding a maximum-length contiguous substring of a given string that is also a palindrome. For example, the longest palindromic substring of "bananas" is "anana". The longest palindromic substring is not guaranteed to be unique; for example, in the string "abracadabra", there is no palindromic substring with length greater than three, but there are two palindromic substrings with length three, namely, "aca" and "ada". In some applications it may be necessary to return all maximal palindromic substrings (that is, all substrings that are themselves palindromes and cannot be extended to larger palindromic substrings) rather than returning only one substring or returning the maximum length of a palindromic substring.

3.5. Enumerating rich strings

The number of distinct subpalindromes in a length n string is at most n. Such strings with exactly n palindromes are called rich. Rich strings possess a number of interesting properties.The sequence A216264 in the Online Encyclopedia of Integer Sequences is the growth function of the language of binary rich strings, i.e., the nth term of this sequence is the number of binary rich strings of length n.

Lemma 1. Any prefix of a rich string is rich.
Proposition 1. Suppose that R is the number of k-ary rich strings of length <= n, for some fixed k and n. Then the trie built from all these strings can be traversed in time O(R).
Proof:
For simplicity, we give the proof for the binary alphabet. The extension to an arbitrary fixed alphabet is straightforward. Consider the following code, using an Eertree on a string with deletions.
void calcRichString(i)
ans[i]++
if (i < n)
if (add('0'))
calcRichString(i + 1)
pop()
if (add('1'))
calcRichString(i + 1)
pop()
Here i is the length of the currently processed rich string. Here, add(c) appends c to the current Eertree and returns the number of new palindromes, which is 0 or 1. Hence the modified string is rich if and only if add returns 1. Note that any added symbol will be deleted back with pop(). So we exit every particular call to calcRichString with the same string as the one we entered this call. As a result, the call calcRichString(0) traverses depth-first the trie of all binary rich strings of length <= n. Here,the pop operation works in constant time. For add we use the method with direct links. Since the alphabet is constant-size, the array directLink[v] can be copied in O(1) time. Hence, add also works in O(1) time. The number of pop's equals the number of add's, and the latter is twice the number of rich strings of length < n. The number of other operations is constant per call of calcRichString, so we have the total O(R) time bound.

4. Applications

1. It can be used to built joint EER tree of several strings.
2. It can be used for computing the number of rich strings up to a given length of a string.
3. It is used to store all palindromic substrings in a string.
4. It can be used to count new palindromic subtrees after insertion in efficiently.
5. It can be used to solve factorization problem in O(kn) time.

5. Conclusions

Previously , When we used to encounter problems based on Palindromes like Maximum length palindrome in a string, number of palindromic substrings and many more such interesting problems , we used to solve them using some DP O(n2) solution (n is length of the given string) or then we have a complex algorithm like Manacher's algorithm which solves the Palindromic problems in linear time.
In this project, we have worked upon an interesting Data Structure - EER TREE, which will solve all the above similar problems in much more simpler way. This data structure was very recently invented by Mikhail Rubinchik and Arseny M. Shur.
An EER tree keeps track of all palindromic substrings of a string in linear time and space. The actual structure of palindromic tree is that of a directed graph rather than a tree. The nodes of a palindromic tree store palindromic substrings.
In this project , We have studied how to build an EER Tree and have analysed its complexity over space and time. We have studied the advantages of EER Tree over other Algorithms used to solve problems based on Palindromes . We have successfully implemented EER Tree to find out all distinct palindromes in input string and longest possible palindrome in the given string.
Possible future developments of this project would be
1. Linear-time factorization into palindromes and an open problem about the optimal construction of the eertree.
2. Optimal construction of the eertree
3. The version of EER Tree that supports deletions from a string.

6. Bibliography and citations

Acknowledgements

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

References

[1] Eertree: An efficient data structure for processing palindromes in strings. *https://www.sciencedirect.com/science/article/pii/S0195669817301294*, 2017.

[2] Alessio Piergiacomi. Eertree (or palindromic tree). Technical report, medium.com, 2016.

[3] Mikhail Rubinchik and Arseny M. Shur. *EERTREE: An Efficient Data Structure for Processing Palindromes in Strings.* PhD thesis, Ural Federal University, Ekaterinburg, Russia, 2014.

[4] Wikipedia. Palindrome tree. Technical report, Wikipedia, 2014.