PALINDROME TREES

August 7, 2024

Utkarsh Patel (2022CSB1140) ,
Vidushi Goyal (2022CSB1142) ,
Umesh Konduru (2022CSB1139)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sravanthi Chede

Summary: Palindrome Tree (or EERTREE) is a linear size data structure that provides fast access to all distinct subpalindromes of a string. In this project, we discuss the complexity analysis of building an EERTREE, and provide an implementation in C. We then discuss further modifications of an EERTREE to support additional operations. We present applications to other algorithmic problems related to palindromes that are efficiently solved using this data structure.

1. Introduction

1.1. Structure of an EERTREE

Edges:
An EERTREE has two types of edges which are as follows:

- Labeled edges: These are directed edges which face downwards. It has a label in the form of a character. If we have a labeled edge from node A to node B with a label c, then B = cAc.
- Suffix links: A suffix edge from a node A will point to a node that contains longest proper suffix of the node that is also a palindrome. Each node has exactly one such suffix link.

Nodes:
In a palindrome tree, each node represents a palindrome, which can be a single character or a longer palindromic substring of the input string. Additionally, the data structure also has two roots which are as follows:

- First root (-1) represents an imaginary string. Following a labeled edge from this root only adds a single character which is in the label of the edge.
- Second root (0) represents an empty string. Labeled edge adds normally on this root.

Note that whenever we add a new character to an existing string, there can be at most one new subpalindrome, i.e., the longest suffix palindrome of the new string. Therefore, a string of length n has at most n distinct subpalindromes. Thus, the tree can have a maximum of n + 2 nodes.

1.2. Building an EERTREE

We design the function add() that adds a new character to the EERTREE of an existing string, while maintaining its properties. We then build the EERTREE online by using repeated calls to the add function.

We initialise the EERTREE by adding two roots: one representing the imaginary string (with length -1) and the other representing the empty string (with length 0).Suffix links for the both the roots point towards the imaginary node.

Consider a call to the add() function. We first need to find the longest suffix palindrome of the new string, and add a node corresponding to it if it doesn't exist. We do so by finding the longest suffix palindrome of the current string that is preceded by the new character c added to the string. Since we know the longest suffix palindrome of the current string from the last call to add, we traverse its suffix links to find the longest suffix palindrome preceded by c. Once we find this palindrome, we search its edges for an outgoing edge labelled c. If it exists, the node corresponding to the new longest suffix palindrome already exists and we're done. Otherwise, we add the new node and continue traversing the suffix palindromes until we find the second longest suffix palindrome preceded by c. This is is the suffix link of the new node.

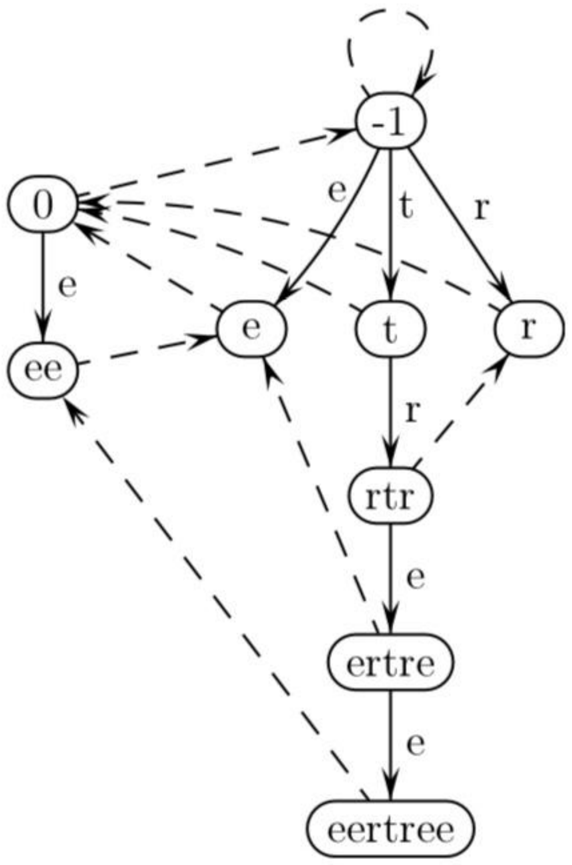

Figure 1: The above figure illustrates the palindrome tree of the string eertree. Labeled edges are solid while suffix links are dashed.

1.3. Complexity Analysis

Time Complexity

Constructing a palindrome tree takes $O(n\log\sigma)$ time, where n is the length of the string and σ denotes the number of distinct symbols in the palindrome. To estimate the time complexity, consider a call to add(). One spends $O(\log\sigma)$ time in searching for the edge labelled c. Therefore, one spends a total of $O(n\log\sigma)$ time in searching for edges. In order to count the number of transitions by suffix links and edges, keep track of the first character of the longest suffix palindrome. A transition by a suffix link moves it to the right, and a transition by an edge moves it exactly 1 character to the left. Since we move by less than n characters to the right, the number of transitions by suffix links is $O(n)$. Therefore, the complexity of building an EERTREE is $O(n\log\sigma)$.

Space Complexity

A palindrome tree takes $O(n)$ space. At most it can take n+2 vertices to store all the sub-palindromes of the string S in the eertree, including n nodes containing sub-palindromes and two roots 0 and -1.

2. Modifications

2.1. Joint EERTREE

Creating a joint eertree can be helpful for handling problems involving multiple strings. Some basic applications include finding number of palindromes common to k given strings, finding the longest suffix palindrome contained in all strings, etc. In fact, they are also useful when we make EERTREEs persistent, in which case we use a joint EERTREE to store different versions of the same string.

Building a Joint EERTREE

To build a joint EERTREE, add the first string as usual. Then set maxsuf (longest suffix palindrome) back to 0, and add the next string. Use a boolean array flag in each node (or any other appropriate data structure, such as a hashtable or a binary search tree) to keep track of the strings in which a subpalindrome is present.

2.2. Speeding up calls to add()

Although an EERTREE can be built in $O(n\log\sigma)$ time, a single call to add() may take up to $O(n)$ time. This may be a hindrance if we would like to support the EERTREE with additional operations such as deletion, or make it persistent. In this section, we discuss a way of speeding up to add() so that it takes $O(\log(n))$ time, with $O(n)$ additional space.

Building an EERTREE using quick links

In order to speed up calls to add(), we store one extra piece of information in each node, namely quicklink. The quick link of a node is the longest suffix palindrome preceded by a character other than the one that precedes link[v]. Computing the quick link of a node v takes constant time. The two longest suffix palindromes of v are link[v] and link[link[v]]. So, quicklink[v] is either link[link[v]] or quicklink[link[v]] depending on the preceding character of link[link[v]].

A typical call to add() traverses the suffix palindromes of a string to find a suffix palindrome preceded by a character c. Using quicklinks, we can reduce the number of suffix palindromes examined. While traversing the suffix links, we check the preceding character of both v and link[v]. If neither of them is c, we can skip to quicklink[v].

3. Applications of EERTREE

The k-factorization asks whether a string can be written as a product of k palindromes, given a k. For example, “malayalam” is 5-factorizable - “m” + “ala” + “y” + “ala” + “m”. An online algorithm that runs in *O(kn)* time was previously known. But using EERTREE, the problem can be solved in time independent of k. In this project, we present an *O(n.log(n))* solution to this problem.
Before solving the k-factorization problem, we look at a very similar problem, namely the palindromic length problem. The palindromic length problem asks for the minimum k such that k-factorization of a string is possible.

3.1. Palindromic Length

The idea is to maintain an array ans[0. . . n - 1], such that ans[i] stores the palindromic length of the substring S[0. . . i] of S. We compute the values of ans online as we build the EERTREE.

First, observe that a k-factorization of S[0. . . i] can be obtained by adding a suffix palindrome P to a k-1-factorization of S[0..i - len(P)]. Therefore, the following holds : ans[i] = min{ans[j] | S[j. . . i]} is a palindrome. Using this relation, we can design a naive approach to compute the ans[i] for each i. Just iterate over the suffix palindromes every time a new character is added, and find the suffix palindrome that gives the minimum palindromic length. However, by this approach, computing ans[n] takes $O(n)$ time, and the overall algorithm takes $O(n^2)$ time.

To speed up this process, we store two extra pieces of information in each node v, namely diff[v] and serieslink[v]. diff[v] = len[v] - len[link[v]], i.e., the difference between the length of the palindrome and the length of its longest suffix palindrome. serieslink[v] is the longest suffix palindrome of v such that diff[v] not equal to serieslink[v]. Computing both diff[v] and serieslink[v] takes $O(1)$ extra time, as serieslink[v] is either link[v], or serieslink[link[v]], depending on whether diff[v] = diff[link[v]]

Let us define a series to be the sequence of all suffix palindromes having the same series link.So a series would look like v, link[v], link[link[v]],...., serieslink[v], where v is included and serieslink[v] is excluded. We call a suffix palindrome to be a leading palindrome of a string if it is the longest suffix palindrome, or if is equal to serieslink[v] for some v. So a series starts with a leading palindrome, and goes until the next leading palindrome, excluding it. The idea is that at any point, the number of leading palindromes is $O(log(n))$, which we shall prove later. Therefore, if we design a way to compute the suffix palindrome that gives minimum palindromic length within a series in $O(1)$ time, iterating through all the suffix palindromes will take $O(log(n))$ time.

Rewriting in terms of series links, computing ans[n] looks like this:

```
 for(Node *v = eertree->max_suf; v->len > 0; v = v->series_link) {
        ans[n] = min(ans[n], getMin(v));
 }
```

where getMin(v) gives the minimum value of ans in the series lead by v. Next, we show how we can use the fact that diif[u] is the same for all u belonging to a series to compute getMin(v) in $O(1)$ time.
diff[v] steps before, we already computed the minimum value of ans in the series of link[v]. Therefore, if we store a parameter pal_len in each node of the eertree, updating it everytime the node becomes a longest suffix palindrome, we can compute getMin(v) as follows.

```
getMin(Node *v) {
        v->pal_len = ans[i - (v->series_link->len + v->diff)) + 1;
        if(v->diff == v->link->diff) v->pal_len = min(v->pal_len, v->link->pal_len);
        return v->pal_len;
}
```

Now, all that remains is to prove that in any path consisting of series links, there are $O(log(n))$ elements.

PROOF
First note that $diff[link[v]] \leq diff[v]$.Indeed, let $v = v[0..m-1]$ be a subpalindrome of S,$t = diff[v], u = link[v]$. If $t \geq \frac{m}{2}$, then $diff[u] < |u| \leq t$,as desired. so assume $t < \frac{m}{2}$. We have $u = v[t..m-1]$ by definition and $u = v[0..m-t-1]$ since v is a palindrome.Hence the string $z = v[2t..m-1]$ equals $v[t..m-t-1]$ as a suffix of u. Now the fact that z is both a prefix and a suffix of the palindrome u implies that z is a palindrome. Therefore $diff[u] \leq |u| - |z| = t$.
Assume that $u = serieslink[v']$ and $u = link[v]$ for a suffix $v = v[0..m-1]$ of v'(possibly v=v').From the definition of series link and the previous paragraph we have $s = diff[u] < diff[v] = t$.Let $z = link[v]$.Then $z = v[t+s..m-1] = v[s..m-t-1]$ as a suffix of u. Since the string $v[s..m-1]$ is not a palindrome by the definition of link but has the palindrome z as its prefix and suffix,the statement "If a palindrome v of length $l \geq \frac{n}{2}$ is both a prefix and a suffix of a string S[0..n-1],then S is a palindrome"implies $m-t-1 \leq t+s$.Hence $m \leq 3t$ and .Thus, every series link transition shrinks the length of a string multiplicatively, hence the result.

3.2. k-Factorization

Given any k-factorization of a string, we can obtain a k+2-factorization by breaking any non-trivial palindrome in the factorization into 3 palindromes (by treating the first and last character as separate palindromes). Therefore, in order to determine the k-factorizability of a string, we need to determine the minimum number of even palindromes and minimum number of odd palindromes it can be broken into. We do this by making a slight modification to the algorithm provided for the palindromic length problem. Instead of maintaining one array, we maintain 2 arrays ans-even and ans-odd. We also maintain 2 additional parameters in each node, pal-len-even and and pal-len-odd. We use the value of pal-len-even to update ans-odd, and value of pal-len-odd to update the value of ans-even.

4. Persistence

In section 2, we presented some modifications of the EERTREE. Among these, we discussed speeding up the add() function, and hinted at the possibility of making EERTREEs persistent. In this discussion we discuss a method to further speed up the add() function, following which we address the problem of making EERTREEs persistent. This section is short, and we only discuss the ideas involved, without going into the details.

4.1. EERTREE using Direct Links

Continuing the discussion on speeding up the add() function, we now look at a method that makes it possible to traverse suffix links in constant time, thereby making add() take $O(\log\sigma)$ time. Instead of storing just one quicklink, we store a binary search tree of direct links, such that directlink[v][c] is the longest suffix palindrome of v preceded by c. Since obtain directlink[v] involves adding at most one node to directlink[link[v]], it is redundant to use a separate binary search tree in each node. Instead, using a fully persistent binary search tree is space efficient, where each version corresponds to the binary search tree of a single node.

4.2. Making EERTREEs Persistent

We also explored the idea of making EERTREEs fully persistent, providing access to all versions of the EERTREE, and making it possible to modify any version of the EERTREE. We have a version tree T, with each node labelled by a character. The node v represents the string read from root to that node. We build a joint eertree for all versions using the method with direct links. Instead of storing a flag array, we store all the subpalindromes of the version v in a binary search tree, which is again fully persistent. We do not go into further details in this project, but it is possible to implement the function addVersion(v, c) in $O(\log(|v|)$ time, where v is the version which we're modifying.

5. Conclusion

In our project we discussed why we initially needed EERTREEs. We also made some useful modifications to it after it's implementation. Then we discussed some real world problems that can be solved using this data structure. We took a brief look at the idea of persistent data structures, and how they can be applied to EERTREEs.

6. Bibliography

1. EERTREE : An efficient data structure for processing palindromes in string - Mikhail Rubinchik, Arseny M. Shur - `https://www.sciencedirect.com/science/article/pii/S0195669817301294`
2. Pal^k is Linear Recognizable Online - Dimitry Kosolobov, Mikhail Rubinchik, Arseny M. Shur - `https://arxiv.org/abs/1404.5244`
3. Making data structures persistent - J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan - `https://www.sciencedirect.com/science/article/pii/0022000089900342`