Purely functional palindromic trees

Fast palindromic analysis in purely functional setting

Nikolai Kudasov
Innopolis University
n.kudasov@innopolis.ru

Timur Khazhiev
Innopolis University
t.khazhiev@innopolis.ru

Abstract

We introduce a purely functional version of palindromic tree (also known as EERTREE), data structure for palindromic analysis on strings. We show that this version does not require users to sacrifice space and time complexity. Furthermore, we introduce a double-ended variation with *merge* operation and show that it can be performed in $O(1)$ on average, openning door for various optimisations, including fast parallel palindromic analysis.

CCS Concepts • **Theory of computation → Data structures design and analysis**;

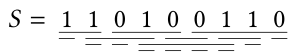
Keywords palidrome, eertree, purely functional, persistent

1 Introduction

A string S is a *palindrome* if it is equal to its reverse: $S = S^{-1}$. For instance, *racecar* is a palindrome, but *runner* is not since $runner^{-1} = rennur$ and $runner \neq rennur$.

Palindromes arise in many research areas, from formal language theory to molecular biology. A lot of solutions to practical problems (such as …) rely fundamentally on some method of efficiently finding all palindromes in a given string.

We will visualise substrings using underlines. For instance, we can visualise all palindromes of length 2 in $S = 110100110$ like this: $\underline{1\,1}\,0\,1\,\underline{0\,0}\,\underline{1\,1}\,0$. By layering these lines we can compactly visualise all palindromes in S:



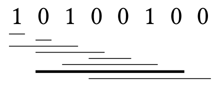
While there can be up to $O(n^2)$ different occurrences of palindromic substrings, it is known [] that there is at most n different palindromic substrings in a string of length n. For instance, $S = 110100110$ has only 8 distinct palindromes[1].

This property suggests existence of linear-time algorithms for palindromic analysis of strings. Indeed, Manacher's algorithm [] is well-known for finding the longest palindromic substring in linear time. Rubinchik's *palindromic tree* (or *eertree*) [1] is a data structure that keeps track of all distinct palindromes in a string and can be built online in linear time.

The latter has a big advantage in that it can be applied to many problems with slight modifications as Rubinchik shows in his paper.

[1]subpalindromes of 110100110 are 1, 11, 0, 101, 010, 00, 1001 and 0110

Manacher's algorithm [] is well-known for finding the longest palindromic substring in linear time. It relies on a simple fact that the longest palindromic substring will also be the maximum palindromic suffix for some prefix of the string as well as a clever way of finding the next maximum suffix in constant time. We can visualise maximum palindromic suffixes for every prefix of a string (longest palindromic substring is visualised with a bold line):



Rubinchik [1] came up with a *palindromic tree* (or *EERTREE*) that is based on similar ideas and allows to collect all palindromes online and also in linear time. The big advantage of this data structure is that it can be applied to various practical problems with slight modifications as Rubinchik shows in his paper.

In his work Rubinchik also introduced a *persistent eertree*, that allows working with multiple versions of a string with little extra cost. His persistent eertree introduces a logarithmic slowdown and can be built in $O(n \log n)$ time. We should note however that his version is not thread-safe and cannot be directly translated to purely functional setting as it relies on a global mutable data structure (joint eertree for all existing versions).

Compared to earlier work, our new contributions are these:

- We introduce an efficient purely functional version of palindromic tree, and provide an implementation in Haskell, a non-strict, purely-functional programming language; we also prove that although adding symbols is $O(\log n)$ worst case, it is $\Theta(1)$ on average.
- We provide a variation of that data structure that is more friendly to garbage collection, and demonstrate its efficiency by counting rich strings up to length n in $O(n)$ memory;
- We design a double-ended variation with new *merge* operation, and show that *merge* can be implemented with $O(1)$ time and space complexity on average.

2 Finding palindromes

Many of the problems that involve palindromes rely essentially on finding all palindromes in a given sequence, possibly keeping track of some extra information (such as number and indices of occurrences).

To the best of our knowledge state of the art for this are Manacher's algorithm and Rubinchik's EERTREE. Both allow processing a string in $O(n)$ with a difference that Rubinchik's approach allows online processing and is more general.

Let $Pal(S)$ be the set of all occurrences of all palindromes in a string S:

$$Pal(S) = \{\langle p, i\rangle | S_{i..j} = p, isPalindrome(p)\}$$

Note that by storing a palindrome occurrence together with its index makes it unique (for a given string) and allows us to perform further analysis. For instance, we can compute total number of occurrences for each palindrome and find a refren-palindrome.

...

3 Palindromic tree

3.1 Naïve palindromic tree

3.2 Infinite palindromic tree

3.3 First applications

4 Memory efficiency

4.1 Counting rich strings

5 Double-ended palindromic tree

5.1 Appending symbols

Here we mention that prefixes and suffixes in palindromes are the same, so we adjust data structure to keep track of both without much changes and no additional asymptotic complexity.

5.2 Merging

As has been shown by Rubinchik and Manacher, it is possible to process a string in linear time. Above we have shown that a similar result can be achieved in a purely functional setting. But all mentioned approaches are inherently sequential and do not allow for an obvious parallelisation to speed up processing large strings.

Our idea and, perhaps, main contribution of this paper is design of a merge operation that allows us to process two strings separately and combine results efficiently.

$$merge(S_1, S_2) = Pals(S_1 + S_2)$$

The idea behind implementation of merge is very simple: we append (or prepend) symbols from one string to another until we know we will not encounter any new palindromes.

To arrive at this implementation and prove its correctness we need first to formalise a few observations.

We define $NewPals(S_1, S_2)$ to be the set of new palindrome occurences, formed after catenation of S_1 and S_2:

$$NewPals(S_1, S_2) = Pals(S_1 + S_2) - Pals(S_1) - Pals(S_2)$$

First observation is trivial, but lets us focus only on the important part of the strings.

Lemma 5.1. *For any two strings S_1 and S_2 every new palindrome occurence $\langle p, i\rangle \in NewPals(S_1, S_2)$ will always cover the catenation point:*

$$i < \|S_1\| \leq i + \|p\|$$

Proof. To be done by Timur Khazhiev. □

Lemma 5.1 hints that we can implement merge by prepending (or appending) symbols, but stop prematurely! The next lemma helps us figure out when exactly we can stop.

Lemma 5.2. *For any two strings S_1 and S_2 every new palindrome occurence $\langle p, i\rangle \in NewPals(S_1, S_2)$ will have its center between centers of maximum palindrome suffix of S_1 and maximum palindrome prefix of S_2:*

$$\|S_1\| - \frac{\|maxSuff(S_1)\|}{2} \leq i + \frac{\|p\|}{2} \leq \|S_1\| + \frac{\|maxPref(S_2)\|}{2}$$

Proof. To be done by Timur Khazhiev. □

Can we stop after new maximum palindome suffix (prefix) has its center outside the interval mentioned in Lemma 5.2? This final lemma helps answer that question:

Lemma 5.3. *Let S' be the result of appending (prepending) symbol c to string S. Then its maximum palindrome suffix center either stays in place or moves to the right (to the left):*

$$\|S\| - \frac{\|maxSuff(S)\|}{2} \leq \|S'\| - \frac{\|maxSuff(S')\|}{2}$$

Proof. To be done by Timur Khazhiev. □

Now all ingredients are in place to implement *merge* that runs linearly in *(number of suffixes/prefixes we need to update?).*

Theorem 5.4. *There is an algorithm that implements merge in $\Theta(K)$, where K is the maximum index in S_2 (correspondingly S_1) corresponding to the rightmost (leftmost) of new palindrome occurrences $NewPals(S_1, S_2)$.* This formulation should be rewritten more clearly.

Proof. To be done by Timur Khazhiev. □

Corollary 5.5. *There is an algorithm that implements $merge(S_1, S_2)$ in $O(min(\|S_1\|, \|S_2\|))$ worst-case.*

Proof. To be done by Timur Khazhiev. □

As we can see, *merge* can be implemented as simple update to the continuous appending (prepending) with virtually no overhead. In fact, since our implementation allows for early exit, it performs better than mere appending for many strings. For example, ...

Interestingly, *merge* will exit early most of the time. More precisely, it will exit in constant time on average.

Theorem 5.6. *merge is $O(1)$ on average.*

Proof. *To be done by Timur Khazhiev. Hint: use theorem about average length of a maximum palindrome suffix.* □

6 Complexity Analysis

Working with maximum palindromic suffixes and prefixes is central to most operations with palindromic trees. So to understand complexities of these operations we study length of such suffixes and prefixes first.

Lemma 6.1. *A random string of length n is a palindrome with probability $\sigma^{-\lfloor \frac{n}{2} \rfloor}$.*

Proof. For a random string to be a palindrome its first $\lfloor \frac{n}{2} \rfloor$ symbols must match symbols from the right half exactly, leaving only 1 valid candidate out of $\sigma^{\lfloor \frac{n}{2} \rfloor}$. □

Theorem 6.2. *Average length of the maximum palindromic suffix of a random string of length n is $\Theta(1)$ for alphabets of size $\sigma \geq 2$.*

Proof. Let $L_{avg}(n)$ be the average length of the maximum palindromic suffix of a random string of length n. $L_{avg}(n) \geq 0$, so to prove $L_{avg}(n)$ is $\Theta(1)$ we need to find a constant upper bound.

Let $P_{max}(k, n)$ be the probability that a string of length n has maximum palindromic suffix of length k. Then $L_{avg}(n)$ can be expressed as a weighted sum:

$$L_{avg}(n) = \sum_{k=1}^{n} k \cdot P_{max}(k, n)$$

We notice that for a string to have a maximum palindromic suffix of length k it is necessary that last k symbols form a palindrome. Using Lemma 6.1 together with that observation we get an upper bound of $P_{max}(k, n)$:

$$P_{max}(k, n) \leq \sigma^{-\lfloor \frac{k}{2} \rfloor}$$

This allows us to derive an upper bound for $L_{avg}(n)$:

$$\begin{aligned} L_{avg}(n) &\leq \sum_{k=1}^{n} k \cdot \sigma^{-\lfloor \frac{k}{2} \rfloor} \leq \sum_{k=1}^{n} k \cdot \sigma^{-\frac{k-1}{2}} \leq \sum_{k=1}^{\infty} k \cdot \sigma^{-\frac{k-1}{2}} \\ &= \frac{\sigma}{(\sqrt{\sigma} - 1)^2} = O(1) \end{aligned}$$

□

Corollary 6.3. $L_{avg}(n) < 12$ *for* $\sigma = 2$.

Corollary 6.4. $L_{avg}(n) \leq 4$ *for* $\sigma = 4$.

Theorem 6.5. *Adding a symbol to the end of the eertree of a string of length n is $\Theta(1)$ on average, but $O(\log n)$ worst case.*

Proof. *To be done by Timur Khazhiev.* □

7 Related and future work

8 Conclusion

References

[1] Mikhail Rubinchik and Arseny M. Shur. 2018. EERTREE: An efficient data structure for processing palindromes in strings. *European Journal of Combinatorics* 68 (2018), 249 – 265. https://doi.org/10.1016/j.ejc.2017.07.021 Combinatorial Algorithms, Dedicated to the Memory of Mirka Miller.