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 $O(\sqrt{n})$ on average, openning door for various optimisations, including fast parallel build of an EERTREE.

CCS CONCEPTS

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

KEYWORDS

palidrome, eertree, purely functional, persistent

1 INTRODUCTION

A palindrome is a string that reads the same both ways. Palindromic patterns appear in many research areas, from formal language theory to molecular biology.

There are a lot of papers introducing algorithms and data structures to facilitate different problems that involve palindromes. One such data structure is EERTREE, a recently described linear-sized palindromic tree introduced by Rubinchik [3].

In this project we aim to design at least one purely functional and fully persistent version of a palindromic tree, implement it in Haskell programming language and compare it with other existing solutions. We are going to start with a naive implementation and gradually arrive at an efficient version, relying on some of the techniques described by Okasaki [1] for designing purely functional data structures.

We hope that purely functional variations will prove valuable for some divide-and-conquer approaches to palindromic analysis. We also believe that a fully persistent version might be useful for comparative analysis of closely-related strings (such as RNA string mutations).

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 of length from 1 to n in $O(n)$ memory;
- We design a double-ended variation with new *prepend* and *merge* operations, and show that *merge* can be implemented with $O(\sqrt{n})$ time and space on average;

2 PROBLEM: PALINDROMIC ANALYSIS

Here we should describe the problem that we are trying to solve. Collecting all palindromes for further analysis based on problem.

3 PALINDROMIC TREE

3.1 Naïve palindromic tree

Structure contains two tries for odd and even palindromes, current maximum palindromic suffix and sequence before that suffix. To add a new symbol to the tree we generate palindromic suffix candidates for input symbol. If symbol before that suffix and input symbol are same, that means that we found a new palindrome, update max palindromic suffix and add new node to the proper trie. If none of candidates is suitable, symbol simple adds to odd trie after the root.

3.2 Infinite palindromic tree

In the original paper author proposed a persistent version of tree, but it based around global mutable structure. We could also achieve persistency using naive implementation, but in that case we need to copy whole structure since there are cyclic dependencies, that is inefficient. To achieve persistency we need to approach a little bit different.

For some alphabet σ we can simply generate palindromes of length n. The number of palindromes of length n will be $g(n, \sigma)$ where

$$g(n,\sigma) = \begin{cases} \sigma^{\frac{n+1}{2}} & n \textit{ is odd} \\ \sigma^{\frac{n}{2}} & n \textit{ is even} \end{cases}$$

That infinite tree will represent all possible palindromes and their dependencies. That tree, in fact, is variation of joint eertree proposed in original paper. So the number nodes of infinite tree for fixed alphabet will be $\sum_{n=1}^{\infty} g(n,\sigma)$. Here where haskell's lazy evaluation will come in handy. We need to calculate only certain subtree from that tree and it will use $O(n)$ memory.

3.3 First applications

4 MEMORY EFFICIENCY

4.1 Counting rich strings

5 DOUBLE-ENDED PALINDROMIC TREE

5.1 Prepending 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.* As append works with maxSuffixes, the prepend should work with maxPrefixes. Palindrome is symmetric, maximum palindrome suffix of palindrome is maximum palindrome prefix as well. Keeping track of maxPrefix will not have any affect on infinite tree. In main structure we should keep track of maxPrefix as well.

5.2 Merging palindromic trees

Here describe an algorithm for efficient merge.

The basic idea is to find palindromes which were not in mergeable trees and combine with old nodes. The merge is simply appending/prepending with the stop criteria. If length of processed string of left tree is greater than length of processed string of right then the algorithm starts by adding one-by-one symbol from right tree to left one. If the center of new palindrome went beyond maxPrefix of right tree, that palindrome is not new to the right tree, moreover all further palindromes will not. That is a stop criteria. Then simply add all already found palindromes from right tree. If length of processed string of left tree is less than length of processed string of right then the operations are opposite (prepend to right one, keep track of maxSuffix of left one).

Lemma 5.1. *Set of palindromes for $S_1 + S_2$ is a union of palindrome sets for S_1, S_2 and some new palindromes that appear at the catenation point. These new palindromes form consecutive descendants of palindromic suffixes of S_1 or palindromic prefixes of S_2.*

Proof. Naively merge can be done as appending all symbols from S_2 to S_1, new palindromes will be build on top of maxSuffixes of some prefix of $S_2 + S_1$. □

Lemma 5.2. *There is an algorithm that implements merge in $\Theta(K)$, where K is the number of new palindromes in $S_1 + S_2$ compared to palindromes of S_1 and S_2 taken separately.*

Proof. New palindromes can be collected by trying to expand suffixes of S_1 and prefixes of S_2. □

Theorem 5.3. *merge is $O(min(N, M, \sqrt{N + M}))$ on average and $O(min(N, M))$ worst case, where N, M are length of input strings.*

Proof. Worst case is when palindrome is almost whole concatenated string, thus processing whole of one strings is required. Another paper from Rubichik [2] showed that random word of length n contains, on expectation, $\Theta(\sqrt{n})$ distinct palindromic factors, so on expectation, concatenation gives us $\Theta(\sqrt{N + M})$ palindromes. □

Proposition 5.4. *On average are $\Theta(1)$ new palindromes with different centers after merge.*

Proof. Length of suffixes (and prefixes) is constant on average, as new palindromes are based on suffixes and prefixes (suffixes and prefixes are basically the centres) their amount is also constant. □

Proposition 5.5. *It is possible to find all largest new palindromes with different centers after merge in $O(\log n + \log m)$ on average.*

Proof. *To be done.* □

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] Chris Okasaki. 1998. Purely Functional Data Structures. (01 1998). https://doi.org/10.1017/CBO9780511530104

[2] Mikhail Rubinchik and Arseny M Shur. 2016. The number of distinct subpalindromes in random words. *Fundamenta Informaticae* 145, 3 (2016), 371–384.

[3] 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.