Purely functional palindromic trees
Iteration II results

Timur Khazhiev
Innopolis University
t.khazhiev@innopolis.ru

Nikolai Kudasov*
Innopolis University
n.kudasov@innopolis.ru

ABSTRACT

This is the results of BS-3 Project and it contains a general project results so far.

CCS CONCEPTS

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

KEYWORDS

palindrome, eertree, purely functional, persistent

1 NAIVE IMPLEMENTATION

Naive implementation was made as a proof of concept. Process of implementation is pretty straightforward. 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.

To check that it works, we can calculate number of rich palindromes of length n and compare it with actual sequence from OEIS [5]. A sequence of length n is called rich if it contains, as subsequences, exactly n distinct palindromes. Also we will use this method of checking further during project.

Code for that implementation can be found in GitHub repository[1].

2 TWO SIDED TREE

As the original tree [4] holds only maximum palindrome suffix, we can only append a new symbol. But what if we want to add a new symbol at the beginning? To allow that we can add maximum palindrome prefix support. So we might need to keep track of max prefixes, but as palindrome is symmetric, maximum palindrome suffix of palindrome is maximum palindrome prefix as well. Thus we can simply keep information about maximum palindrome prefix of processed string (as we keep information about maximum suffix). So no major memory usage is added, only one more entry.

This extension allows us merge two eertrees more efficient than simple naive approach (adding symbol by symbol from string of eertree_2 to eerteee_1). Let's consider example: "*abbabba*" + "*bbbab*". Palindromes of concatenated string includes all processed palindromes of both strings and new ones. It's pretty obvious that new palindromes can be found somewhere around joint of two strings.

There are three cases: (1) center of new palindromes is joint (2) center of new palindromes is max suffix (and/or its max suffixes) of first string (3) center of new palindromes is max prefix (and/or its max prefixes) of second string. For current example blue represents max palindrome prefix, red represents max palindrome suffix and black represents symbols that we are checking: "*abbabba*" + "*bbbab*"

(1) "*abbabba*" + "*bbbab*", "*ab*" is not a palindrome, stop here.
(2) _"*abbabba*" + "*bbbab*", nothing to compare, moving to next max suffix
"*abbabba*" + "*bbbab*", "*babbab*" is potential new palindrome, expanding more
"*abbabba*"+"*bbbab*", "*bbabbabb*" is potential new palindrome, expanding more
"*abbabba*"+"*bbbab*", "*abbabbabbb*" is not a palindrome, moving to next max suffix.
"*abbabba*" + "*bbbab*", "*bab*" is potential new palindrome, expanding more
"*abbabba*" + "*bbbab*", "*bbabb*" is potential new palindrome, expanding more
"*abbabba*" + "*bbbab*", "*abbabbb*" is not a palindrome, stop here.
(3) "*abbabba*" + "*bbbab*", "*abbba*" is potential new palindrome, expanding more
"*abbabba*" + "*bbbab*", "*babbbab*" is potential new palindrome, expanding more
"*abbabba*" + "*bbbab* ", nothing to compare, moving to next max prefix
"*abbabba*" + "*bbbab*", "*abbb*" is not a palindrome, moving to next max prefix
"*abbabba*" + "*bbbab*", "*abb*" is not a palindrome, stop here

All found palindromes are added to the a tree. Max palindrome suffix of merged tree is max palindrome suffix of second string. If one of new found palindromes includes whole second string, it becomes max palindrome suffix. And visa-versa max palindrome prefix of merged tree is max palindrome prefix of first string. If one of new found palindromes includes whole first string, it becomes max palindrome prefix. Thus max suffix can be same sequence as max prefix.

Another paper from Rubichik [3] showed that random word of length n contains, on expectation, $\Theta(\sqrt{n})$ distinct palindromic factors. So if length of first string is n and length of second is m, on expectation, their concatenation gives us $\Theta(\sqrt{n+m})$ palindromes. From that we exclude palindromes that already processed separately. Since max palindrome suffix and prefix can be new palindromes from concatenation we can update them both at $O(\sqrt{n+m})$.

*Project supervisor.

3 INFINITE 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

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

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.

Unary infinite tree was implemented[2].

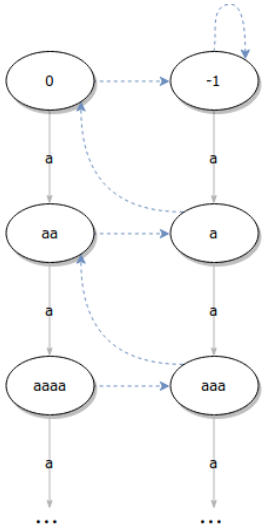

Figure 1: Infinite tree over unary alphabet

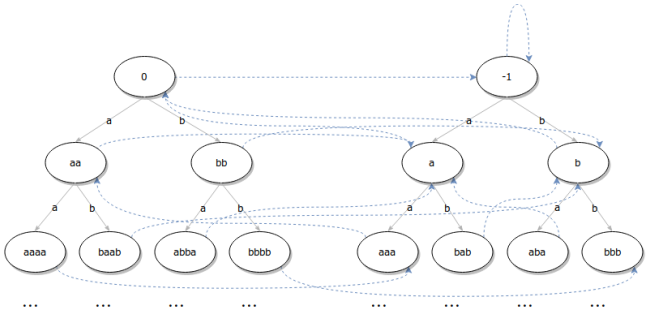

Figure 2: Infinite tree over binary alphabet

REFERENCES

[1] Timur Khazhiev. 2019. Naive implementation. https://github.com/khazhix/ds-project/blob/iteration_ii/implementation/eertree.hs. (2019).
[2] Nikolai Kudasov. 2019. Infinite eetree over unary alphabet. https://gist.github.com/fizruk/6162521c515aea7266cccce28505aabc. (2019).
[3] Mikhail Rubinchik and Arseny M Shur. 2016. The number of distinct subpalindromes in random words. *Fundamenta Informaticae* 145, 3 (2016), 371–384.
[4] 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.
[5] "Jeffrey Shallit". 2013. The On-Line Encycloped Integer Sequences. https://oeis.org/A216264. (Mar 2013). Number of rich binary words of length n.