EERTREE demand, I am a maid named EERTREE*

Brady J. Adcock[a,1], Abdel R. Issa[a,1], Luke T. Patterson[a,2], Hayden M. Fravel[a,2]

[a] *Appalachian State University : 287 Rivers St, Boone, NC 28608, United States*

Abstract

An exploration and implementation of palindromic trees', EERTREEs'[2], whose construction is linear space. This structure provides fast access to all sub-palindromes of a string or set of strings. Some applications include palindromic factorization problems, rich string enumeration, and identification of Watson-Crick (WK) palindromes [1].

Keywords: palindrome, EERTREE, rich string, Watson-Crick palindrome

1. Introduction

This paper is an extension of Mikhail Rubinchik and Arseny M. Shur's paper "EERTREE: An Efficient Data Structure for Processing Palindromes in Strings"[2] which presents EERTREE's to the world. We implemented the structure in C++ with a focus on readability instead of efficiency. We used this implementation to explore EERTREEs' competitive programming applications, a bio-computational application, review the structures performance in the real world, and discuss asymptotic bounds of the enumeration of rich strings using a modified version of the tree.

EERTREEs are a novel, double rooted, and elegant tree structures created for the purpose of finding all sub-palindromes in a provided string. These trees solve the most important palindromic structure problems with linear space and almost linear time: searching for and counting palindromes in a string and factorization of a string into palindromes. The tree can solve even more problems with modification and as such deserve an in-depth analysis.

*This document is the result of many long nights and debilitating anxiety.

Email addresses: adcockbj@appstate.edu (Brady J. Adcock), issaa@appstate.edu (Abdel R. Issa), pattersonlt@appstate.edu (Luke T. Patterson), fravelhm@appstate.edu (Hayden M. Fravel)

[1] I now have aibohphobia.

[2] I now have ailihphilia.

Our Motivation

EERTREEs are very new to the world and yet they are an approachable data structure for students making them a perfect subject for a term paper. The trees are simple to understand once built and their construction is intriguing much like the suffix tries and trees they're inspired by. For these reasons and more we decided to study and extend Rubinchik and Shur's paper.

2. Preliminaries

Palindromes

Before discussing the data structure its important to review palindromes. Any string that reads the same backwards as it does forward is a palindrome, so a palindrome $P = a_1a_2a_3...a_n$ is equal to its reversal $\overleftarrow{P} = a_n...a_2a_1$. Some of the commonly known palindromes are race-car and aibohphobia. Aibohphobia means fear of palindromes! Another example is the word *aabb* which is a special type of palindrome called a rich string.

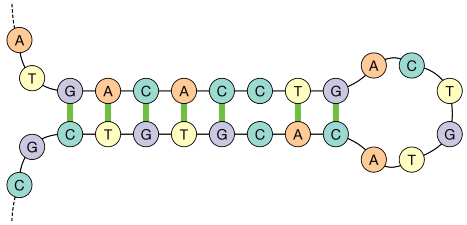
Rich String

Rich strings are palindromes who have n unique sub-palindromes. The EERTREE paper's first lemma allows us to conclude that the maximum number of distinct sub palindromes possible for a given string of length n is n thus rich strings are special. Some more examples include: "010" "000111" "10101". It is very easy to create rich strings when $|\sigma| = 2$ and $n \leq 8$ because every possible word within those bounds is rich. In our extension section we begin discussing the enumeration of rich strings of different alphabet sizes and the asymptotic bounds present.

Bio-computation

DNA is a part of all of us. It is inside every living thing on Earth, which makes it pretty important. DNA is made in the shape of a double helix and has four different bases: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). When looking at the structure of a DNA strand, A always pairs with T, C always pairs with a G and vice versa. These pairings are labeled as Watson-Crick (WK) compliments: T's WK compliment is A, G's WK compliment is C and so on[1]. We can go further and describe an entire single strand of DNA as WK complementary. That is to say that $GGTT$ is WK complementary of $CCAA$.

When attempting to encode information on a single DNA strand it is important to prevent "undesirable interactions" such as the first half of a strand which is WK complimentary of its second half annealing with the second half creating a hairpin structure as seen below.



These structures and more strict Watson-Crick palindrome strands[3] have been correlated to cancer genomes, particularly breast cancer genomes[4]. These structures are the focus of many studies which aim to encode data in DNA and then complete biocomputations followed by decoding because these hairpins render the computations useless. When retrieving the output of a bio-computation a WK complement is usually annealed to the processed strand but if that strand has already annealed to itself the processed data is lost[1].

Watson-Crick Palindrome

A WK palindrome is a DNA strand that is the same when WK complimented and reversed, that is to say WK involuted. For example, the DNA strand $TGCA$ would have a WK compliment of $ACGT$ and the reverse of this compliment is $TGCA$ which is clearly the same as the original strand $TGCA$ making it a WK palindrome. So we can say that if the WK involution of a strand s is equal to s then s is a WK palindrome: $\text{WK}(s) = s$ then s is θ-palindrome. One extension explored in this paper is the use of eertrees to find all WK palindromes in a given genome[4].

Notations

- $S[n]$ means the nth character in some string S.
- $S[i : j]$ defines the sub-string starting at i and ending at j from the string S.
- T denote translation tables which its entries are pairs of characters. T has the property that every character (in the alphabet) only appears once in the first position, and once in the second position. Note that some letter should not map to itself.
- $\text{WK}(s)$ is the WK involution of the DNA strand s.
- θ-palindrome means WK palindrome.
- $u \longrightarrow v$ describes edge from node u to node v.
- $y \Longrightarrow w$ describes suffix link from node y to node w.
- $link[y]$ describes the node the suffix link from y points to.
- ϕ is the null node.
- λ is the empty node.
- $eertree(S)$ for some string S is the tree whose processed string is S.

[3] Many studies look for WK palindromes with some threshold for errors.
[4] https://www.ncbi.nlm.nih.gov/genome?term=vih&cmd=DetailsSearch

- σ represents the alphabet, set of possible characters that can make a word, for some language.

3. Algorithm

A Tree With Two Roots

Below is an overview of eertree components and the *add*() function. This information is all that is needed to understand the simplest form of eertrees.

Node

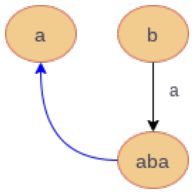
Eertrees' spring from two root nodes, the null node and the empty node. Before discussing these special nodes lets discuss the average palindrome node. Each node contains a sub-palindrome of the processed string. Following are 3 example nodes from the string "aba"[5].

Edge

Each node not only contains a string of the palindrome (when implemented nodes contain just their length and edges are used to construct the strings stored by the nodes) which it represents but also an edge and a suffix link. An edge is directed between nodes, from node u to node v, and describes how to get the palindrome stored in v from the palindrome stored in u by using the the edge label. The label is the character c which gets added to both sides of u to create v. For example, $u =$ "b" with edge $u \longrightarrow v$ that has the label "a" then $v =$ "aba".

Suffix Link

Suffix links describe the longest proper suffix (suffix which is not equal to the entire string) of the origin node which is a palindrome. Put another way, a suffix link from a node y which leads to w, $y \Longrightarrow w$, indicates that w is the longest palindromic proper suffix of y. The following image shows an example suffix link and edge. Note that this is not an eertree but an example of eertree node interactions. (need figure stuffs)



[5]This happens to be a rich string!

Root Nodes

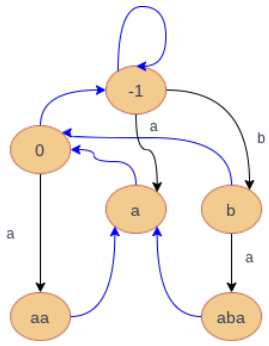
Now that we understand palindrome node attributes we can discuss the two root nodes which make eertrees possible. The first root is the null node, ϕ, which has length of -1. It's suffix link is itself for reasons that will become apparent later. The other root is the empty node, λ, whose length is 0 and suffix link is ϕ. These nodes allow for palindromes of length one and two to have a suffix links and edges which leads to them. Lets see how these short palindrome nodes get created using $add()$.

$add()$

The simplest eertree includes just one method, $add(a)$ for some character a. If the call to $add(a)$ creates a new node in the data structure the method returns 1 otherwise it returns 0. The method also concatenates a to the processed string. Lets consider a call to $add("a")$ for an $eertree("aba")$. This means the processed string, T, is currently $"aba"$ and after the call to $add()$ is complete $T = "abaa"$. We know that our current longest suffix palindrome, Q, is the node $"aba"$ and therefore its length l is 3. To find the new longest suffix palindrome of Ta we traverse the suffix-palindromes starting at Q and for each palindrome read its length l and compare $T[i-l]$ where i is the last index of T to the character being inserted, $"a"$.

So in our example: $T[2-3]$ is not a valid character and therefore we traverse the suffix link of Q to the node $"a"$. This means we check $T[2-1]$ for equality to $"a"$ which also isn't a valid character so we traverse again. This time we land on λ and so we check the character at $T[2-0]$ which is $"a"$! This means we need to check λ for edges whose label is $"a"$ because if that edge exists then we know this palindrome is a duplicate and all that needs to be done is updating our current longest suffix palindrome. In our exmaple we create a new node $"aa"$ because λ does not have any edge labeled $"a"$. $\lambda \longrightarrow "aa"$ is created because of our definition of edges.

It remains to create the suffix link for our new node. This is done in the same way as above, begin at Q and traverse the suffix links until an equality is reached. If you're doing this by hand its easiest to just remember that the suffix link should always lead to the second longest suffix-palindrome of Ta. Below is $eertree("abaa")$:



Assuming "a" had been a unique character that had not yet been processed then we would have traversed the suffix links until ϕ which means that we would check ϕ for edges labeled by "a" and created/updated longest suffix palindrome node accordingly. The suffix link of "a" in this case would have been λ by definition.

This basic version of add takes $O(1)$ space and is lower bounded by σ but unfortunately this $add()$ is $O(n)$ run-time for one call. The original paper walks through some modifications which allow for much much better bounds for one call to $add()$ and we implemented the quickLinks which is one of those modifications. The fastest $add()$ that we know of is done using "directLinks" which have $\Theta(\log \sigma)$ time for one call to $add()$. This requires an increase in the space for each node to $O(min(\log \sigma, \log^2 n))$. Over n calls though all types of add preform the same on average at $\Theta(n \log \sigma)$. The average time for one call to $add()$ matters most when implementing $pop()$ which is also discussed later.

Modifications

Following is an overview of the modifications we were able to implement and their applications. This includes joint eertrees, $pop()$, and quick links.

Joint Eertrees

Joint eertree is an eertree that is able to hold multiple strings. $eertree(S_1, S_2, ..., S_k)$ is built as follows:

1. Build $eertree(S_1)$
2. Set the maximum suffix to be 0
3. Repeat on other strings S_j

To make this modified tree useful we modify the nodes to contain a list of the strings which created the given node. With this modification it becomes trivial to compute the occurrence count of any given palindrome within the set of input strings and solve the palindromic refrain problem. This modification also modifies the run-time which becomes $O(kn \log(\sigma))$, and the space for each node becomes $O(k)$.

Pop

Pop can be implemented by using a stack which stores the longest suffix palindrome of each call to $add()$. When $pop()$ is called we need to erase the node that is being pop'd assuming it is not a duplicate. $pop()$ can be implemented in constant time as we'll discuss in our extension.

Quick Links

For the observant reader, they may notice a problem with our current understanding of add and pop. Namely, consider the following scenario:

$$add(a), ..., add(a), add(b), pop(), add(b), pop(), ..., add(b).$$

If we do $O(n)$ calls to $add(a)$, then each $add(b)$ will be $O(n)$, thus the entire sequence will be $\Omega(n^2)$, regardless of how *pop* is written. To solve this, we can use quick links to make a single call of $add()$ take only $O(\log(n))$ time.

Consider some node v, its suffix link $link[v]$, and the symbol $b = v[len(v) - len(link[v])]$. We define the quick link of v to be the longest suffix palindrome of v, preceded in v, by a symbol different from b.

This helps reduce the running time as follows, if the current palindrome is v and we are adding in c, and if both v and $link[v]$ are not preceded by c, then all suffix palindromes longer than the quick link of v will not be preceded by c. Thus, we skip them, and check the quick link directly.

4. Extensions

Implementation

We have implemented this data structure in C++, with the following extensions:

- Ability to handle more than one string
- Ability to use quick links
- Ability to pop

The code can be found on github here. It is possible to extend this code to many other applications discussed in the paper, but we did not have time to implement some of the major applications.

Joint Eertrees

Our code to implement joint trees follows exactly as stated earlier in the algorithm section on joint trees, build the tree for the first string in the set of strings to be processed. Then set the maximum suffix to be the λ and then call build() for next string in the set of inputs. Repeat this until all input strings have been processed.

Quick Links

Our implementation of Quick links uses an unordered map, keyed on the labels of the nodes, as the core container for all the links. For every newly added palindrome v, we check the following condition: $S[n - len(link(v))] == S[n - len(link(link(v)))]$. If so, then $QL[v] = QL[link[v]]$, otherwise, $QL[v] = link[link[v]]$. This is only $O(1)$ work, thus the time complexity of $add()$ is $O(\log(n))$. These implementations follow from how quick links are defined.

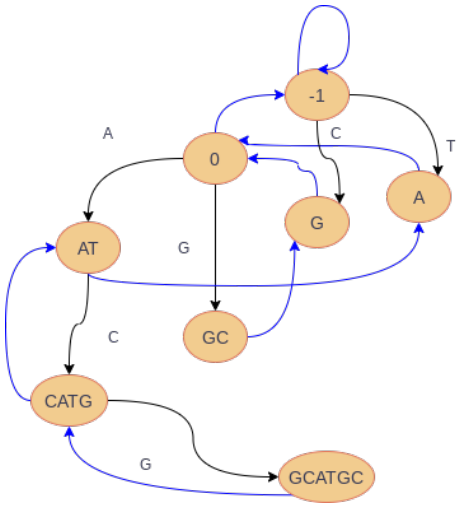
pop()

We implemented pop in $O(1)$ by introducing a stack into the data structure that keeps track of the previous list of longest suffix nodes. For every insertion, push onto the stack the pair $\langle int, bool \rangle$, where *bool* refers if the current pair was a result of a new palindrome, or a duplicate one. And, *int* is the current longest palindrome suffix node index. If *bool* is false, then it is simple to go back a state, by just setting the longest palindrome suffix node index to whatever was stored in the pair. This is simply $O(1)$ time. If *bool* is true, then it is more complex to go back a state, since we not only need to set the longest palindrome suffix node index to whatever was stored in the pair. We also need to erase the node that is being removed, which was just created, from essentially everywhere. However, since we are using unordered maps to store nodes, this can be done in $O(1)$ time. Therefore, *pop* is simply a $O(1)$ operation.

Finding WK Palindromes

We attempted to use eertrees to find all WK palindromes in the X genome of a common fruit fly. We explored the idea by changing the character that the suffix palindrome traverse function searched for to be the WK compliment of that character instead. This wasn't easily implemented in our code because we use the suffix traverse function to compute the longest palindrome suffix node for new nodes as well as next nodes. Our rough sketches show that we should be able to get this working theoretically but we ran out of time. WK palindromes can not be odd length and they can not be empty[1].

The following algorithm was used to create this modified WK eertree: begin the WK eertree with the same λ and ϕ nodes. When inserting a new node and creating an edge, instead of labeling the created edge with the character being inserted we instead label the edge with the WK compliment of the character being added. We will still store the entire WK palindrome in the node (just store indices to save space, this is something we did not do and it hurt us) as with the original eertree. Following is an example of the WK eertree for the input DNA strand $GCATGC$.

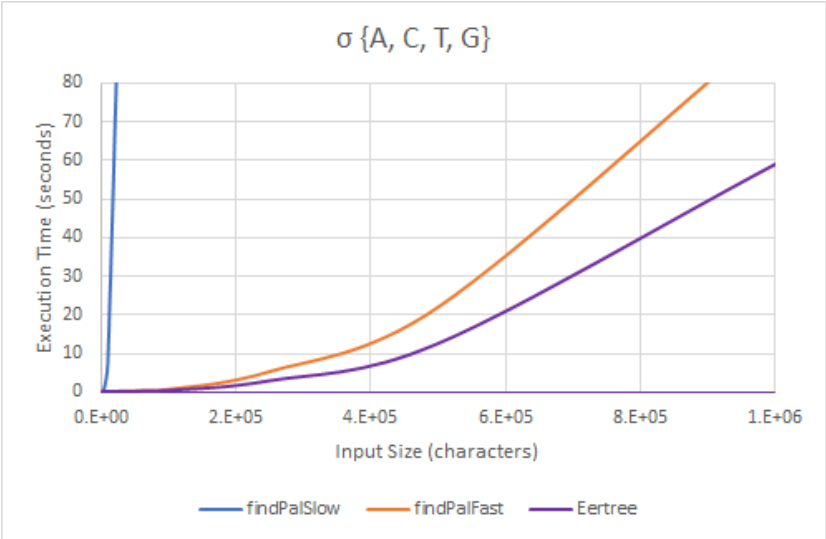


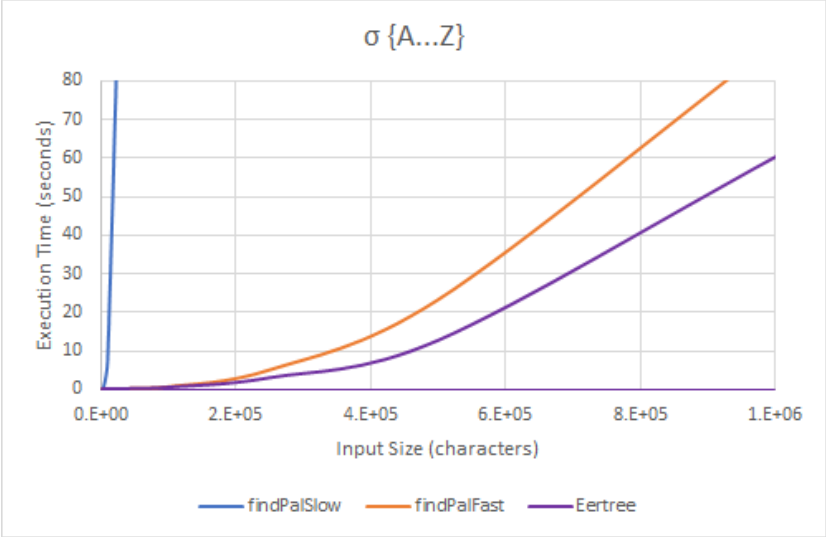
Because WK palindromes can not be odd length and they can not be empty we must ignore the odd length nodes which the modified eertree create and ignore those nodes which are of length 1 when querying the data structure for valid WK palindromes. We do not have the time to prove that this is a valid way to check for all WK palindromes but because the original paper listed WK palindromes as an application we think it is possible in a manner that is at least similar to this if not this exact strategy. We're also unsure whether or not quickLinks and or directLinks work to speed up this WK eertree idea. We think these links should work but we also think the average runtime for add will change to $\Theta(n)$

Running Time

Following is an overview of the time improvements that eertrees provide over naive algorithms for finding all sub-palindromes of random strings with varying lengths over varying alphabets. We also go over run-time of rich string generation for varying alphabet sizes and maximum lengths.

The following graphs show the runtime difference between $findPalSlow()$, an $O(n^3)$ solution to finding all sub palindromes in a given string, $findPalFast()$, an $O(n^2)$ solution with a very small constant, and $eertree()$ which is $O(n \log(\sigma))$. Our observed running time is similar for $\sigma = 4$ and $\sigma = 26$. However, there should be a significant increase in the running times. We think this is due cache misses caused by the storing of the entire strings each node holds instead of storing just indices which define the string with the node stores. This also caused our algorithm to use over 120GB of memory to compute the rich string for $\sigma = 3$ and $n = 25$.





Rich Strings

Theorem 1. *We can generate all strings from an alphabet σ, and of length n using translation tables, T, and starting at a fixed letter.*

Proof. We can use $\sigma - 1$ translation tables to be able to generate all the strings of length n. We start by assuming, without loss of generality, all strings start

with the first letter of the alphabet, lets call it a, and call the last letter of the

alphabet z. Start by generating all of the σ^{n-1} strings of length n that start with an a, we will call this set of strings A, we will call the set of all strings of length $n-1$ B. Then, start with generating a table T_1 as follows: $a \to b$, $b \to c$, ..., $z \to a$. Then, define $T_i = T_1$. Notice that after applying the table $i-1$ times, the letter a will shift to z. So, if we apply the translation table one more time, we will get a duplicate. After we apply the tables T_1, T_2, ..., $T_{\sigma-1}$, to the set we will have generated all possible strings of length n. Notice that applying T_i to the elements of B still gives us B. Namely, consider a string $b \in B$, then there must exist a string $b' \in B$ such that after applying T_i to b', we get b, and b will replace some element $b'' \in B$, otherwise B wouldn't contain all the strings of length $n-1$. Thus, for every element in B it will replace an element, and be replaced by some element after applying T_i. Assume that we have a missing string X, assume that X starts with the j^{th} character in σ, then X must be present after applying $T_1, T_2, ..., T_j$ to the elements of A. Since, after applying all these transformations, the initial letter of every string in A is the j^{th} character in σ, and therefore $X \in A$ after the transformation, since the transformation is closed on B. This concludes the proof.

Theorem 2. *The translation of a rich string is rich*

PROOF. We can show this to be true by using induction on the length of the string. If the length of the string is 1, then the case is trivially true, since all strings of length 1 are rich. If the length of the string in n, then we can consider the two following cases:

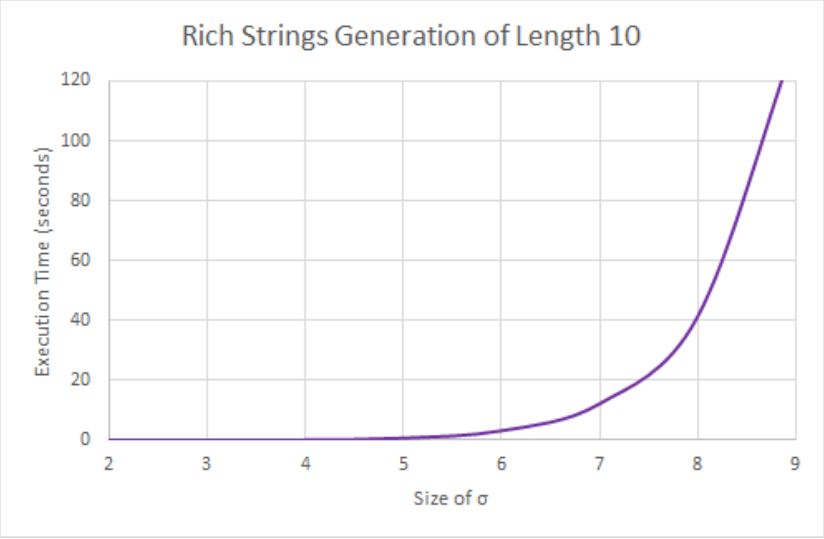
- $S[n] \notin S[: n-1]$ In this case, regardless of what the new translation will result in for $S[n]$, we know that $S[n]$ after the translation is not in $S[: n-1]$ after the translation. Thus, the new string generated after the translation must be rich.
- $S[n] \in S[: n-1]$ In this case, we can try to see this using a dummy letter !, that will replace all occurrences of $S[n]$ in S before the transformation. After applying the transformation, to the first $n-1$ letters, we still have a rich string. Thus, when adding the ! back, we are getting back the original string S, however, we now have removed some letter from S (Namely, if

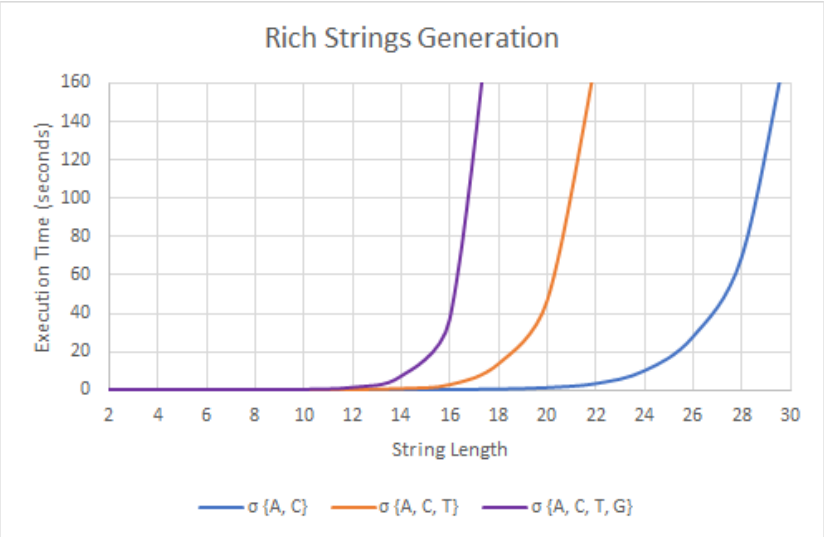
$S[n]$ mapped to j, then j is not present in S anymore). From here, we can replace ! with the missing letter, and we will get a rich string. Since replacing some character with an unused character will not change the fact that the string is rich.

This concludes the proof.

With these two theorems in mind, we are able to generate all the rich strings in a σ order of magnitude lower than the brute force method. Regardless, the number of strings we have to go through is σ^{n-1}, which is still an exponential function. Since our current method of doing $add(c)$ is $O(\log(n))$ and $pop()$ is $O(1)$, this means that we expect this to be $O(\sigma^{n-1}\log(n))$ in terms of time complexity.

Further research on the topic of bounds was interesting. It appears to be that for binary strings, the upper bound on the number of rich strings from some length n is $O\left(\frac{n}{g(n)}\right)^{\sqrt{n}}$, where $g(n)$ is some infinitely growing function[3]. It would be interesting to find an upper bound that depends on σ and n, however that appears to be not discussed. We have included below a couple of graphs that show the execution time needed to generate rich strings under various different conditions.





We have also started the process for submitting a couple of the results, namely all the ones we could generate without crashing our computers for $\sigma < 5$, to The On-Line Encyclopedia of Integer Sequences (OEIS).

5. Conclusion

An eertree is an elegant data structure whose uses are slowly being recognized by the world. They are able to find all sub-palindromes in a string and create a tree in only linear space and almost linear time. This has been shown to have possible benefits in the competitive programming world and even the bioinformatics world.

References

[1] Lila Kari and Kalpana Mahalingam. Watson–crick palindromes in dna computing. *Natural Computing: An International Journal*, 9(2):297, 2010.

[2] Mikhail Rubinchik and Arseny M. Shur. Eertree: An efficient data structure for processing palindromes in strings. *European Journal of Combinatorics*, 68:249 – 265, 2018.

[3] Josef Rukavicka. Palindromic factorization of rich words. 2021.

[4] Sandeep Subramanian, Srilakshmi Chaparala, Viji Avali, and Madhavi K. Ganapathiraju. A pilot study on the prevalence of dna palindromes in breast cancer genomes. *BMC Medical Genomics*, 9:251 – 259, 2016.