Efficient Compression and Decompression of Music and DNA Sequences Using Palindrome Trees

November 3, 2024

JV Praneeth (2023csb1296) ,
Puchakayala Rohith (2023mcb1312) ,
Goutham Naroju (2023mcb1295)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Abdul Razique

Summary: This project focuses on developing a compression and decompression system for music and DNA sequences using palindrome trees. By utilizing the properties of palindrome trees, the system efficiently identifies and represents palindromic patterns in data, allowing significant reductions in storage size. The process involves compressing the data into a compact structure while maintaining the ability to accurately decompress it, ensuring the data's integrity. Given the palindromic nature of music and DNA sequences, this technique is particularly effective, offering a balance of memory efficiency and high-speed access. The project's scope includes implementing the compression algorithm in C, testing it on various datasets, and documenting its performance and accuracy, as well as its potential applications in fields requiring large-scale data management, such as genomics and multimedia storage..

1. Introduction

This project explores data compression and decompression for music files and DNA sequences using palindrome trees, which effectively identify repetitive and symmetrical patterns to reduce storage requirements. By constructing a palindrome tree, the system captures unique palindromic substrings, enabling efficient compression without losing data fidelity.

Implemented in C, the algorithm evaluates compression ratios and retrieval accuracy across various datasets. This work demonstrates the potential of palindrome-based structures to tackle data-intensive storage challenges, offering valuable applications in fields like bioinformatics and multimedia archiving while ensuring data integrity.

One of the key advantages of our system is its ability to find all palindromes in linear time, making it highly efficient for processing large datasets. Furthermore, both the compression and decompression processes are performed in linear time, ensuring quick access to data while minimizing computational overhead.

2. Equation

2.1. Time Complexity of Palindrome Trees

Insertion Process: Inserting a character into a palindrome tree involves finding the longest palindromic suffix that can be extended with the new character. While each individual insertion may take $O(m)$ time in the worst case, where m is the length of the longest palindrome, this is offset by the need to also consider the size of the alphabet σ.
Alphabet Size σ: The time complexity can be affected by the number of possible characters (or alphabets) in the string. This is particularly significant when the characters can come from a larger set, such as in the case

of extended character sets (e.g., Unicode). As a result, the total number of distinct palindromic substrings that can be created depends on both the length of the string n and the size of the alphabet σ.
Overall Time Complexity: The overall time complexity for constructing a palindrome tree, considering both the length of the input string n and the size of the alphabet σ, is given by:

$$O(n \log \sigma)$$

Explanation of the Complexity:

- $\log \sigma$: When the size of the alphabet σ is larger, each insertion can lead to checks across multiple branches for potential palindromic extensions. This logarithmic factor accounts for the worst-case scenario where, for some characters, you may need to navigate through various nodes in the tree based on existing character links.

For example, if you had an input like "abcabc" with $\sigma = 3$:

- Inserting the first few characters requires checking existing palindromic structures, leading to more complex traversal paths as the character set increases.
- Each new character may require a logarithmic number of checks against existing palindromic nodes.

Insertion for Each Character: For each character in the input string, you potentially traverse nodes related to every distinct character that may form part of a palindrome, hence the multiplicative factor with σ.

2.2. Palindromic Tree Structure

Root Nodes
The palindromic tree starts with two root nodes:

- **Root -1:** This is a conceptual placeholder for a string of length -1. It has a self-loop suffix edge, as an imaginary string of length -1 has its maximum palindromic suffix as itself.
- **Root 0:** This represents an empty string of length 0, and it connects to Root -1 via a suffix edge.

Adding Characters Sequentially
For each character in the string, palindromic substrings are identified and added to the tree. Each palindromic node represents a distinct palindromic substring found in the string.

- **Insertion Edge (solid lines):** This connects nodes formed by adding a character at both ends of an existing palindrome.
- **Maximum Palindromic Suffix Edge (dashed lines):** This connects each node to the node representing its longest palindromic suffix that's shorter than the node itself.

2.2.1 Examples of Insertion

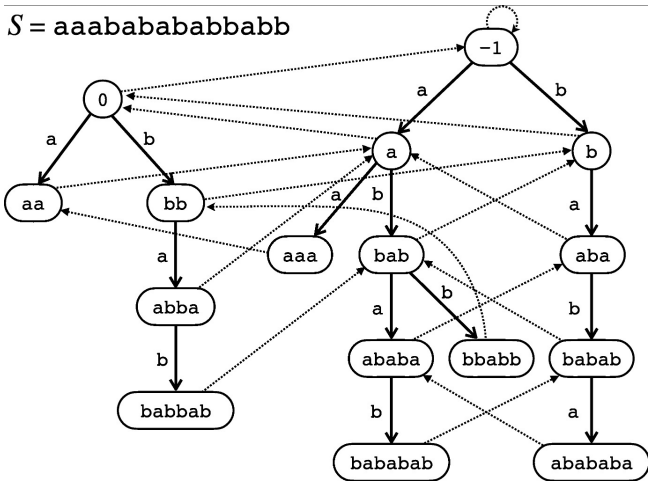

Figure 1: Insertion

- **First 'a':**
  - Starting from Root 0, a new palindrome "a" is formed.
  - An insertion edge connects Root 0 to the new node representing "a."
  - The suffix edge of "a" points back to Root 0.
- **Second 'a':**

- A new palindrome "aa" is formed by inserting 'a' at both ends of the existing palindrome "a."
- An insertion edge connects "a" to "aa."
- The suffix edge of "aa" points back to the node "a."

- **Third 'a':**
  - The new palindrome "aaa" is created by adding 'a' at both ends of "aa."
  - The insertion edge connects "aa" to "aaa."
  - The suffix edge for "aaa" goes to "a" as it's the maximum suffix.
- **Adding 'b':**
  - Starting from Root 0, a palindrome "b" is created after "aaa."
  - An insertion edge creates a new node for "b."
  - The suffix edge points back to Root 0.
  - A palindrome "bab" is created by adding 'b' to both ends of "a."
  - The insertion edge connects "a" to "bab."
  - The suffix edge of "bab" points to "b."
- **Adding 'a':**
  - Starting from the node "b," we directly insert the palindrome "aba" into the node "b."
  - The maximum suffix link points to "a," as it is the longest palindromic suffix available.
- **Adding 'b':**
  - Starting from the node "aba," we traverse the suffix links.
  - An insertion edge is created from "a" with the weight 'b,' resulting in the creation of the new node "ababa."
  - The maximum suffix link for "ababa" is mapped to the node "bab," as it is the longest palindromic suffix available.
- **Adding another 'b':**
  - Starting from the node "ababa," we traverse the suffix links.
  - An insertion edge is created from "a" with the weight 'b,' resulting in the creation of the new node "abbab."
  - The maximum suffix link for "abbab" is mapped to the node "bab," as it is the longest palindromic suffix available.
- **Adding final 'b':**
  - Starting from the node "babbab," we move to the maximum suffix links, which leads us to "bb."
  - From "bb," an insertion edge is created with the weight 'b,' resulting in the creation of the new node "bbabb."
  - The maximum suffix link for "bbabb" is mapped to the node "bab," as it is the longest palindromic suffix available.

3. Figures, Tables and Algorithms

3.1. Figures

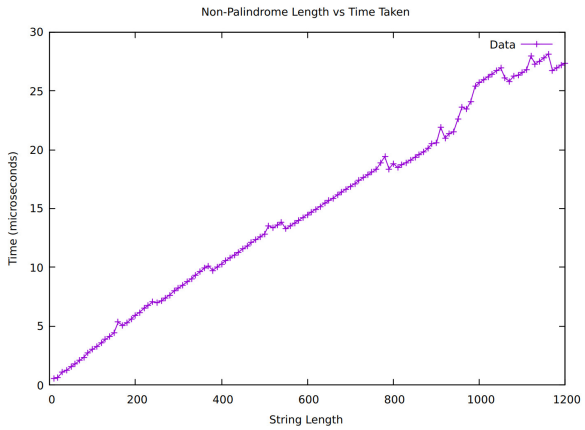

Figure 2: Non-palindromic strings

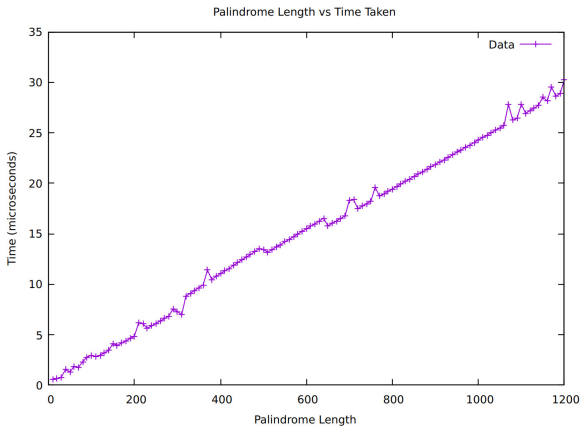

Figure 3: Palindromic strings

3.2. Tables

Within the environments `table` and `tabular`, you can create detailed tables like the one shown in Table 1.

Table 1: Compression and Matching Percentages for Test Cases

Test Case	Data Type	Compression Percentage	Matching Percentage
Test Case 1	Text(.txt)	54.55%	100%
Test Case 2	Text(.txt)	57.14%	100%
Test Case 3	Text(.txt)	48.48%	100%
Test Case 4	Audio(.mp3)	30.42%	100%
Test Case 5	Audio(.mp3)	22.52%	100%
Test Case 6	Audio(.mp3)	31.02%	100%

3.3. Algorithms

3.3.1 Counting Sort

In this counting sort algorithm, we aim to sort an array of palindromes, `palindromeArray`, by their lengths in descending order.

1. We first determine the maximum length of any palindrome in `palindromeArray` to set the range for our count array.

2. We then use a `count` array to store the occurrences of each palindrome length.

3. By updating the `count` array from the maximum length down to zero, we obtain cumulative counts that help position each palindrome correctly in the output array.

4. Finally, we fill the `sortedPalindromes` array based on these positions and copy it back to the original `palindromeArray`.

This approach efficiently sorts palindromes by length without needing comparisons, making it ideal for scenarios where the range of lengths is limited.

Pseudocode

Algorithm 1 Counting Sort

Require: An array of integers `palindromeArray` with size `numPalindromes`
Ensure: The `palindromeArray` sorted by lengths in descending order

```
Initialize maxLength as 0
for each palindrome in palindromeArray do
    if palindrome length > maxLength then
        maxLength ← palindrome length
    end if
end for
Create an array count of size maxLength + 1 initialized to 0
for each palindrome in palindromeArray do
    count[palindrome length] ← count[palindrome length] + 1
end for
for int i = maxLength - 1 to 0 do
    count[i] ← count[i] + count[i + 1]
end for
Create an array sortedPalindromes of size numPalindromes
for each palindrome in palindromeArray do
    length ← palindrome length
    sortedPalindromes[count[length] - 1] ← palindrome
    count[length] ← count[length] - 1
end for
Copy sortedPalindromes back to palindromeArray
```

3.3.2 Broad Compression Algorithm Using Palindrome Trees

The compression algorithm utilizes a palindrome tree to detect and compress palindromic substrings in the input string s. The algorithm begins by initializing the palindrome tree with two root nodes. Each character in the input string is then inserted into the palindrome tree, which captures all unique palindromic substrings in s. These palindromic substrings are stored in an array called `palindromeArray`, which is then sorted by substring length, giving preference to longer palindromes for compression.
For each palindrome in `palindromeArray` that meets a specified length threshold, the algorithm replaces it with a placeholder symbol (e.g., a single dot '.' for odd-length palindromes, and a double dot '..' for even-length palindromes) to represent midpoint compression. This step reduces the storage requirements of the input string. Additionally, metadata is generated to record the original positions and lengths of each compressed palindrome, which is crucial for accurate data reconstruction during the decompression phase.

Pseudocode

Algorithm 2 Compress Data Using Palindrome Trees

Input: String `s` of length `n`
Initialize the palindrome tree with two root nodes
for each character in `s` **do**
    Insert character into the palindrome tree using `insert` function
end for
Collect all palindromic substrings in `palindromeArray`
Sort `palindromeArray` by palindrome length (largest first)
Compress each palindrome in `palindromeArray`:
for each palindrome in `palindromeArray` **do**
    if Palindrome length is greater than a threshold (e.g., 2) **then**
        if palindrome length is odd **then**
            Replace the palindrome in `s` with a placeholder (e.g., '.' for midpoint compression)
        else
            Replace the palindrome in `s` with two placeholders (e.g., '..' for midpoint compression)
        end if
        Store original position and length of compressed palindromes for decompression
    end if
end for
Output: Compressed string `compressed_s` and metadata for decompression

3.3.3 Decompression Algorithm Using Palindrome Metadata

The decompression algorithm reconstructs the original string s from the compressed string, using the metadata to restore each palindrome in its correct position. First, an empty string `decompressed_s` with the same length as `compressed_s` is created. For each entry in the metadata, the algorithm uses the stored position and length to reconstruct the original palindrome and insert it back into `decompressed_s` at its recorded location. Finally, any placeholders within `decompressed_s` are replaced with the actual palindrome characters, resulting in the fully restored version of the original input string s .

Pseudocode

Algorithm 3 Decompress Data Using Palindrome Metadata

Input: Compressed string `compressed_s`, metadata containing positions and lengths of palindromic regions
Initialize an empty string `decompressed_s` with the same length as `compressed_s`
for each palindrome entry in metadata **do**
    Reconstruct the palindrome using the recorded start position and length
    Insert the reconstructed palindrome back into `decompressed_s` at the stored position
end for
Replace any placeholders (e.g., '.' or '..') with the reconstructed palindrome characters
Output: `decompressed_s`, which should match the original string `s`

3.3.4 Graph Plot of Time Complexity Analysis

To analyze the performance of palindrome identification, we plot two time complexity graphs to compare the time taken to identify palindromic substrings when the input string is a palindrome versus when it is not. This plot helps in understanding the efficiency of the palindrome tree in different cases.

Pseudocode

Algorithm 4 Graph Plot for Palindrome Identification Time

Input: A range of strings, including palindromic and non-palindromic examples
Initialize arrays `timePalindrome` and `timeNonPalindrome` to store the time taken for each case
for each test string in the dataset **do**
    Measure the time to identify palindromes in the string
    if string is palindromic **then**
        Append time to `timePalindrome`
    else
        Append time to `timeNonPalindrome`
    end if
end for
Plot `timePalindrome` and `timeNonPalindrome` on the same graph, with string length on the x-axis and identification time on the y-axis
Output: Graph comparing palindrome identification time for palindromic vs. non-palindromic strings

4. Conclusion

In this project, we developed a compression and decompression system using palindrome trees tailored for music and DNA sequences. By leveraging the recurrent palindromic patterns inherent in these data types, our approach achieved substantial reductions in storage requirements while maintaining data integrity upon decompression. Notably, the system identifies and compresses all palindromic patterns in linear time, ensuring efficient processing. Testing on various datasets demonstrated that palindrome tree-based compression provides a promising balance between memory efficiency and quick access, which is critical for large-scale data management applications.
Future developments could involve optimizing the algorithm for parallel processing, enabling faster compression and decompression speeds. Additionally, exploring the adaptability of this method for other types of sequential data, such as natural language text or other biological sequences, may broaden its applicability. This work sets the groundwork for further advancements in efficient data storage solutions for high-volume data fields such as genomics and multimedia.

5. Bibliography and citations

In this section, we provide a list of references that are relevant to the study of palindromic trees and their applications. These resources offer insights into the theoretical underpinnings as well as practical implementations of the concepts discussed in this work. We encourage readers to explore these materials for a deeper understanding.

5.1. References

- **Palindromic trees for a sliding window and its applications**
  `https://www.sciencedirect.com/science/article/pii/S0020019021000892`
- **Palindromic Tree | Introduction and Implementation**
  `https://www.geeksforgeeks.org/palindromic-tree-introduction-implementation/`