LZW Compression Project Report

Muhammed Enes Gündüz - 150120038

Course: CSE 4077 Advanced Data Structures

Date: 18.12.2025

Project: LZW Encoder-Decoder with Array, Trie, and Patricia Trie

1. Introduction

The objective of this project was to implement a complete LZW (Lempel-Ziv-Welch) compression and decompression system and analyze how different data structures affect its performance. LZW is a dictionary-based algorithm that builds a symbol table dynamically as it reads the input. The efficiency of the algorithm depends almost entirely on how fast we can query and update this dictionary.

In this project, I implemented the dictionary using three different strategies:

1. **Baseline Array:** A simple list using linear search.
2. **Trie (Prefix Tree):** A tree structure where each node represents a character.
3. **Patricia Trie:** A space-optimized trie that compresses single-child chains into merged edges.

This report documents the design choices, implementation details, and experimental results comparing these three methods across different file types.

---

2. Design and Implementation

The core logic of the LZW compressor remains the same across all tasks. The compressor reads a character C, appends it to the current prefix P, and checks if P + C exists in the dictionary. The difference lies solely in *how* we store and search for P + C.

2.1 Task 1: Baseline Array

The first implementation uses a simple array (Python List) to store dictionary entries.

- **Search Strategy:** To check if a string exists, the program iterates through the entire list from index 0 to the end.
- **Complexity:** This is an O(N) operation, where N is the current size of the dictionary. As the dictionary grows (up to 65,536 entries), this linear scan becomes a significant bottleneck.
- **Storage:** We store the full byte string for every entry.

2.2 Task 2: Trie-Based Dictionary

The second implementation replaces the array with a Trie.

- **Structure:** The dictionary is a tree rooted at an empty node. Each node represents a prefix, and its children represent the next possible characters.
- **Search Strategy:** Instead of concatenating strings and scanning a list, we maintain a pointer to the "current node." When a new character C arrives, we simply check if the current node has a child labeled C.
- **Complexity:** This is O(1) relative to the dictionary size (or O(L) where L is the string length). It does not get slower as the dictionary grows.

2.3 Task 3: Patricia Trie

The third implementation uses a Patricia Trie (Radix Tree) to optimize memory.

- **Optimization:** In a standard Trie, a string like "Apple" requires 5 nodes. If "Apple" is the only word starting with "A", the Patricia Trie merges these into a single edge labeled "Apple".
- **Insertion Logic (Edge Splitting):** The most complex part of this implementation was handling insertions. If we have an edge labeled "Apple" and we need to insert "Apply", the system detects the divergence at 'e' vs 'y'. It splits the existing edge into "Appl", creates a branch, and adds two new leaf nodes.

---

3. Pseudocode

The following pseudocode illustrates the core logic used for the Trie-based compressor, which was the most efficient implementation.

```
None

Initialize Trie with ASCII characters (0-255)

CurrentNode = Root.Children[FirstByte]


For each subsequent Byte C in Input:

    If CurrentNode has child C:

        // Match found, extend the prefix

        CurrentNode = CurrentNode.Children[C]
```

```
    Else:
        // No match found
        Output Code(CurrentNode)

        // Add new entry to Dictionary
        Create new Child C for CurrentNode
        Assign Next_Available_Code to New Child

        // Reset search to the new character
        CurrentNode = Root.Children[C]

Output Code(CurrentNode)
```

---

4. Experimental Results

I evaluated all three implementations on four different file types:

1. **Plain English:** Standard text with natural repetition (english.txt).
2. **Genome:** DNA sequences (A, C, G, T) with high randomness but small alphabet (test_genome.txt).
3. **Source Code:** Python code with structural keywords (test_source.txt).
4. **Synthetic:** Highly repetitive data (AAAA...BBBB...) to test maximum compression (test_synthetic.txt).

4.1 Summary Table

Input Type	File Size	Method	Compression Ratio	Speed (MB/s)	Peak Dict Size
English	1.2 KB	Array	0.99	0.05	902
		Trie	0.99	1.51	902
		Patricia	0.99	0.87	902
Genome	1.0 MB	Array	3.59	0.00	65,536
		Trie	3.59	4.12	65,536
		Patricia	3.59	0.43	65,536
Source	8.0 KB	Array	1.47	0.03	2,995
		Trie	1.47	2.28	2,995
		Patricia	1.47	1.10	2,995
Synthetic	1.0 MB	Array	158.71	0.04	3,563
		Trie	158.71	10.58	3,563

		Patricia	158.71	2.44	3,563

---

5. Analysis and Discussion

5.1 The Performance Gap (Array vs. Trie)

The most striking result is seen in the **Genome** test.

- **Array:** 443 seconds (0.00 MB/s) [1]
- **Trie:** 0.24 seconds (4.12 MB/s) [2]

This massive discrepancy confirms the theoretical limitations of the array-based approach. The Genome file filled the dictionary to its maximum capacity (65,536 entries). For the Array implementation, this meant that for *every single input character*, the program had to scan through up to 65,536 items to check for a match. This O(N times M) complexity makes the array method unusable for files larger than a few kilobytes.

In contrast, the Trie implementation performed consistent lookups regardless of the dictionary size, processing the 1MB file in under a quarter of a second.

5.2 Trie vs. Patricia Trie

While the Patricia Trie is more memory-efficient in theory, my experimental results showed it was consistently slower than the standard Trie in terms of throughput (e.g., 4.12 MB/s vs 0.43 MB/s on Genome data).

This is likely due to implementation overhead. In the standard Trie, moving to a child is a simple dictionary lookup. In the Patricia Trie, we must perform string slicing and character-by-character comparisons to verify if an edge label matches the input. For the Genome file, where the alphabet is small (4 chars) and the tree is dense, the overhead of managing complex edges outweighed the benefits of having fewer nodes.

5.3 Compression Ratio

The compression ratio was identical across all three methods, which confirms that the underlying LZW logic was implemented correctly in all three tasks. The data structure changed *how* we stored the data, not *what* data we stored.

- **Best Case:** The Synthetic file achieved a ratio of **158:1**. LZW excels at long runs of identical characters.
- **Genome Surprise:** The Genome file compressed well (3.59 ratio) despite being "random" DNA. This is because the alphabet size is small (4 characters). Storing them as 8-bit ASCII is inefficient; LZW effectively re-encoded them into packed sequences, demonstrating LZW's ability to adapt to the entropy of the source.

6. Conclusion

This project demonstrated the critical importance of selecting the right data structure for dynamic algorithms. While the Array-based dictionary was easiest to implement (Task 1), it failed to scale. The Trie implementation (Task 2) offered the best balance of speed and complexity, providing massive performance gains. The Patricia Trie (Task 3), while conceptually powerful for memory saving, introduced computational overhead that reduced throughput for the specific test cases used.

The final deliverable includes a working compressor/decompressor that can handle any binary input and reliably restore it, verified by MD5/byte-comparison checks.