Deflating DEFLATE: Optimising Compression Algorithms

Jacob Biggs

FALMOUTH UNIVERSITY

Abstract

This project involves the development, profiling, and optimization of the DEFLATE algorithm in C++. Initially, the algorithm was implemented in C++, followed by a detailed profiling to understand its performance. Based on the insights gained from profiling and benchmarking, the software was then optimized to enhance its runtime efficiency. This project provides a comprehensive understanding of the DEFLATE algorithm and the importance of profiling and optimization in software development. Finally it discusses the trade-off between time complexity and compression ratio within different approaches to DEFLATE.

DEFLATE Algorithm

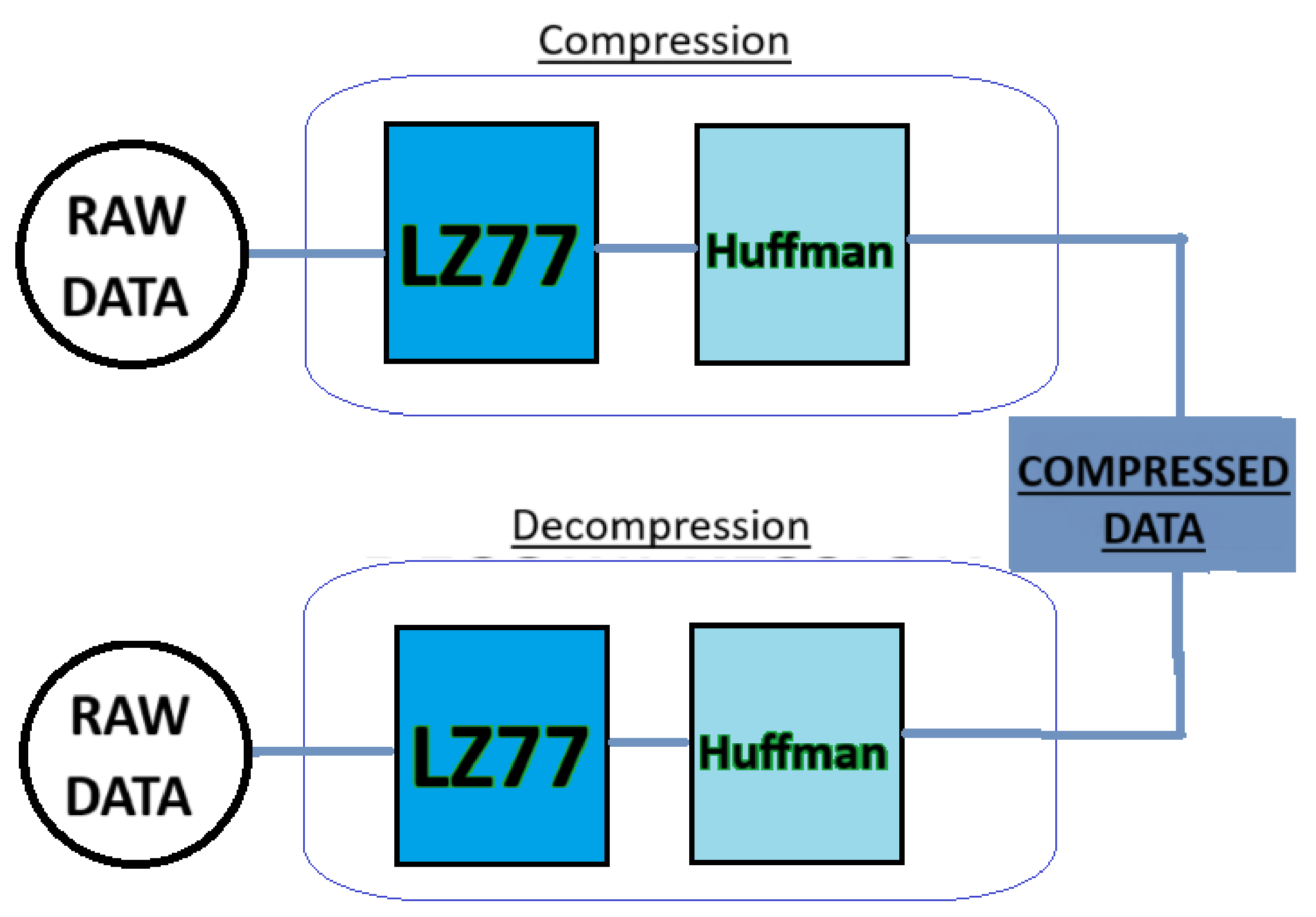

Figure 1. DEFLATE/INFLATE Process Diagram

The DEFLATE algorithm, a lossless data compression algorithm, is a combination of the LZ77 algorithm and Huffman coding. The primary objective of this project is to understand the intricacies of these algorithms, implement them in a constrained environment, and optimize the implementation for better performance. The DEFLATE algorithm is widely used in various applications, including the gzip file format and the PNG image file format [1].

Lempel-Ziv vs Huffman Compression

Lempel-Ziv Encoding

LZ77 Compression, a variation of Lempel-Ziv, is a lossless compression algorithm that replaces occurrences of data with a reference to an earlier encountered copy. It uses a "sliding window" approach that combines a dictionary of repeated substrings within the data, and a look-ahead buffer to encode [1] [2]. The algorithm outputs a sequence of tokens representing either matches found, or literals if no match was found. Each token consists of an offset, length, and the character (or byte) following the match. If no match was found, a token with offset and length 0 is created, which can inflate data if not carefully managed [2]. The time complexity of LZ77 is $O(n^2)$, but can be improved with sophisticated data structures [3].

Huffman Encoding

Huffman encoding is a another technique for lossless data compression. It assigns shorter variable-length binary codes to more frequently occurring bytes (or characters) and longer codes to less frequently occurring ones [4] [1]. The algorithm involves building a Huffman tree where each leaf node corresponds to a byte within the data. The code for that byte is found by traversing the Huffman tree from the root to the desired leaf node [5]. The power of Huffman coding comes from the unique prefixes, meaning no two codes will be made up of the same prefix. The time complexity of building the Huffman tree is $O(nlogn)$, where n is the number of unique bytes/characters, rather than the length of the data [6].

Comparison

The choice between the two often depends on the nature of the data being compressed. For example, LZ77 can be more effective for compressing data with many repeated substrings, while Huffman encoding can be more effective for compressing data where certain characters occur very frequently [7] [8]

Baseline Profiling

First the LZ77 algorithm was implemented. Each distance, length, next char triple was encoded into an "LZ77 Token" struct. The time complexity of this implementation is $O(n^2)$ because for each position in the input, the algorithm may need to search the entire look-ahead buffer. More efficient implementations of LZ77 can achieve a time complexity of $O(nlogn)$ using suffix trees/arrays [3]. The Huffman Coding algorithm was implemented by counting the frequencies of each byte, and then creating tree nodes with matching frequencies. When building a Huffman Tree, the two bytes with lowest frequencies must be repeatedly extracted. Initially this was done with a std::sort function, which gave an overall time complexity of $O(n^2logn)$. This is inefficient as the list must be sorted $n-1$ times in a worst case scenario. After the initial implementation, the algorithm was tested and ran on both Windows and Linux (Ubuntu). This was profiled and benchmarked on both Perf profiler, and Google Benchmark by compressing the bee movie script 100 times over. This helps better understand where the most time was spent, and where improvements are needed.

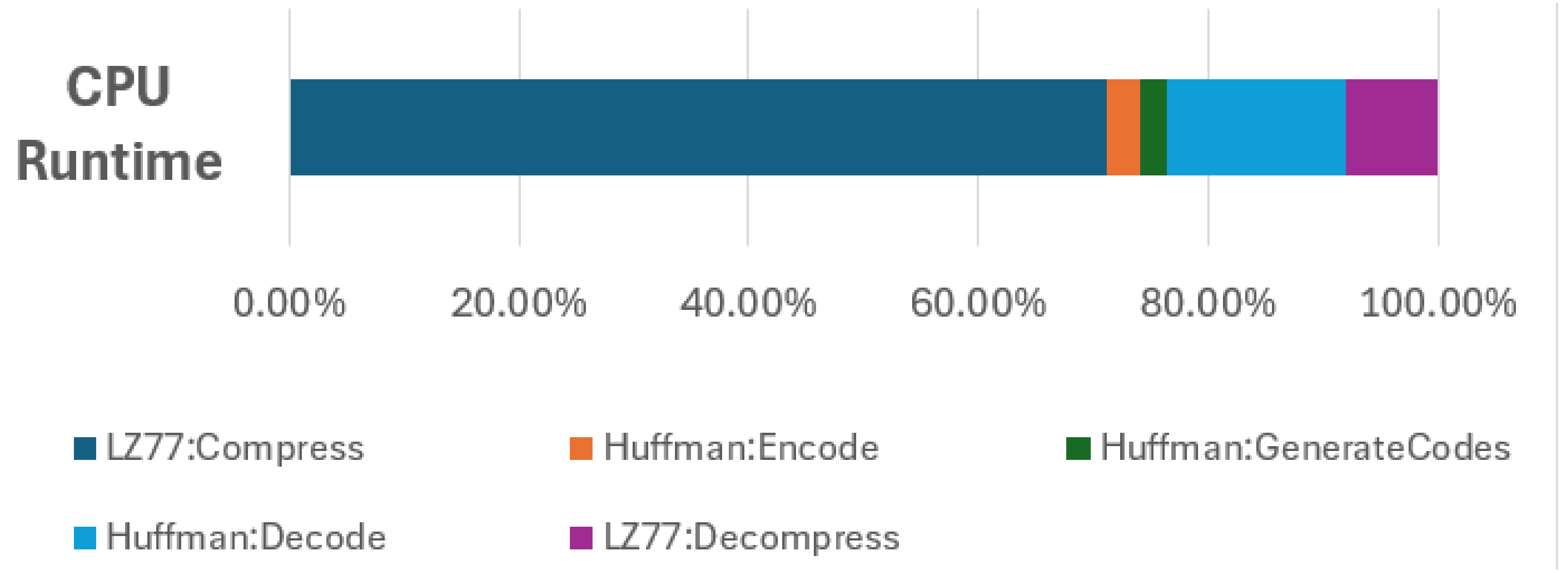

Figure 2. Percentage of runtime taken, data from perf flamegraph.

1. **LZ77 Linear Search:** Although great at finding the longest matches, the linear search implemented into the Lempel-Ziv compression was a big bottleneck for the algorithm
2. **Huffman Decoding:** The huffman decoding process was taking second longest. The use of sophisticated data structures will help improve this.
3. **Maintaining Comrpession Ratio:** Often, the price of faster runtime speed comes at the cost of compression ratio, particularly with LZ77, due to it operating with a search algorithm. In the following optimisations, compression ratio will try to be maintained across all changes.

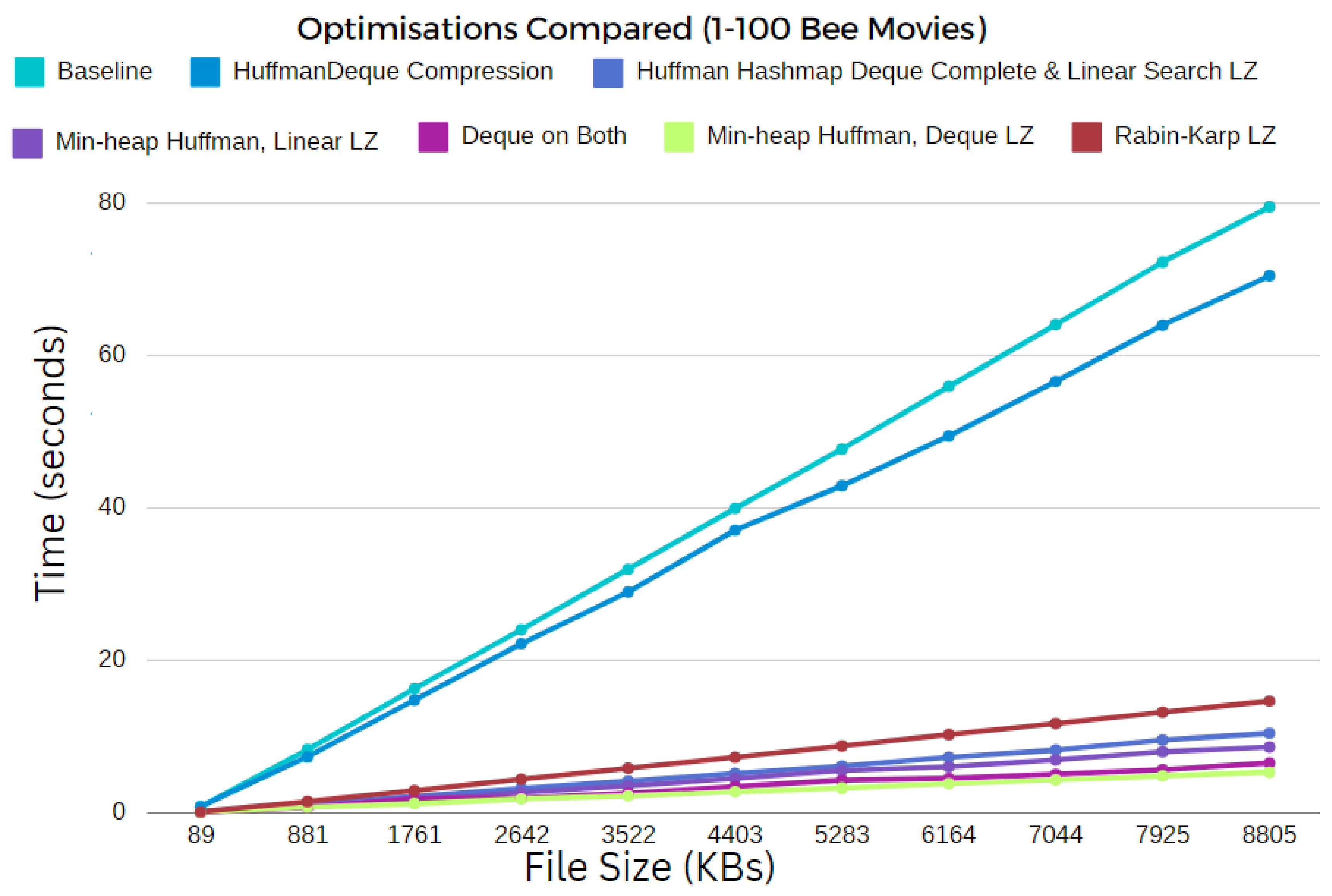

Figure 3. Optimisations Compared on 1-100 Bee Movies

Optimisations Analysis

During optimisation, various alternative approaches were tested. For Lempel-Ziv, a classic approach is to use a rolling hash function, such as Rabin-Karp [9]. Although theoretically, the Rabin-Karp allows for constant time insertions, in practise, the worst-case time complexity of this implementation was still $O(n^2)$ with linear search outperforming this in some contexts due to it not being utilised for multiple patterns. In addition to this, suffix arrays combined with a binary search was also tested, but found to reduce the compression ratio, due to the divide-and-conquer nature of a binary search. When implemented with a more dynamic dictionary and look-ahead buffer, this optimisation can achieve compression in $O(nlog(m))$ time, where n is the length of the data, and m is the size of the dictionary.

Huffman: Min-heap and Trie

A min-heap is a complete binary tree where the value of each node is less than or equal to the values of its children. In the context of Huffman coding, a min-heap is used to efficiently find the two nodes with the smallest frequencies when building the Huffman tree. By using a min-heap, the Huffman coding algorithm builds the Huffman tree in O(n log n) which is a great improvement from $O(n^2logn)$.

The Trie allows the decoding process to be done in O(n) time, where n is the size of the encoded data. This is because each bit in the encoded data corresponds to a single step in the trie, and the time to take a step in the trie is constant, regardless of the size of the data or the number of unique bytes. This makes the trie a very efficient data structure for the Huffman decoding process.

Lempel-Ziv: Double Ended Queue

In the deque optimization version, the algorithm uses a hash map to quickly find potential matches in the sliding window. Each entry in the hash map is a deque that stores the positions of a particular character in the window. This optimization allows the algorithm to quickly find potential matches in the window without having to iterate over all the characters. The time complexity of this operation is O(1) on average. Therefore, the overall time complexity of the deque optimization version is O(n), where n is the size of the input data. This was a great improvement over the $O(n^2)$ linear search.

Recommendations/Future Work

It is evident that the combination of Huffman encoding with a Min-heap and decoding with a Trie, along with the Deque version of LZ77, provides an efficient solution for data compression and decompression with DEFLATE. When tested compared to baseline on the largest file size tested, the compression sped up 12.66 times and the decompression 83 times. This combination leverages the strengths of both algorithms, providing a balance between compression ratio and time complexity. Huffman encoding, with its use of a Min-heap and Trie, offers efficient encoding and decoding processes, making it a suitable choice for applications where data transmission speed is crucial, such as packets over a network. On the other hand, the Deque version of LZ77, with its efficient search mechanism, provides a robust solution for compressing data with repeated substrings. However, it is important to note that the performance of these algorithms can be influenced by the nature of the data being compressed. Therefore, it is recommended to consider the characteristics of the data when choosing a compression algorithm.

References

[1] S. Oswal, A. Singh, and K. Kumari, "Deflate compression algorithm," *International Journal of Engineering Research and General Science*, vol. 4, no. 1, pp. 430–436, 2016.

[2] A. D. Wyner and J. Ziv, "The sliding-window lempel-ziv algorithm is asymptotically optimal," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 872–877, 1994.

[3] A. J. Ferreira, A. L. Oliveira, and M. A. Figueiredo, "Suffix arrays-a competitive choice for fast lempel-ziv compressions," in *International Conference on Security and Cryptography*, vol. 2, pp. 5–12, SCITEPRESS, 2008.

[4] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.

[5] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, vol. 6, no. 2, pp. 163–180, 1985.

[6] R. Hashemian, "Memory efficient and high-speed search huffman coding," *IEEE Transactions on communications*, vol. 43, no. 10, pp. 2576–2581, 1995.

[7] A. Singh, B. S. Khehra, and G. K. Kohli, "Differential huffman coding approach for lossless compression of medical images," in *Intelligent Computing and Communication: Proceedings of 3rd ICICC 2019, Bangalore 3*, pp. 579–589, Springer, 2020.

[8] B. I. Diop, A. D. Gueye, and A. Diop, "Comparative study between different algorithms of data compression and decompression techniques," in *Proceedings of the International Conference on Paradigms of Computing, Communication and Data Sciences: PCCDS 2022*, pp. 737–744, Springer, 2023.

[9] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka, "Approximating lz77 via small-space multiple-pattern matching," in *Algorithms-ESA 2015: 23rd*