INF 242: Compression of the Folktale Using LZW and Huffman Encoding

Paal Halvorsen

September 27, 2024

Contents

1 Introduction

The objective of this project is to implement two compression algorithms—Lempel-Ziv-Welch (LZW) and Huffman encoding—to compress the folktale *Askeladden som stjal sølvendene til trollet*. The program must apply LZW compression followed by Huffman encoding to further reduce the size of the compressed data. The compression rates for each stage are reported and analyzed.

2 Lempel-Ziv-Welch (LZW) Compression

2.1 Overview of LZW Compression

Lempel-Ziv-Welch (LZW) is a dictionary-based lossless data compression algorithm. It builds a dictionary of substrings dynamically during the encoding process and replaces repetitive occurrences of substrings with shorter codes. LZW starts with an initial dictionary containing all individual characters of the input alphabet. As new substrings are encountered, they are added to the dictionary and are represented by a unique code.

2.2 Implementation in Python

The LZW algorithm was implemented in Python, both for compression and decompression. Below are the key steps involved:

- **Dictionary Initialization:** The dictionary is initialized with the 29 Norwegian letters and a space, assigning each a unique code.
- **Compression:** The input text is parsed character by character, and substrings are matched against the dictionary. When a substring is found that is not in the dictionary, it is added, and the previous substring is output as its corresponding code.
- **Decompression:** The decompression process reconstructs the original text from the sequence of codes using the dictionary.

The compression rate achieved for the folktale file using LZW alone was 0.6578, meaning LZW compressed the original text to approximately 65.78% of its original size.

2.3 Decompression Validation

The decompression function was tested to ensure that it accurately restored the original text. By comparing the decompressed output with the original folktale, we validated that no data was lost or altered during compression and decompression.

3 Huffman Encoding Integration

3.1 Overview of Huffman Encoding

Huffman encoding is another lossless compression algorithm, which works by assigning variable-length codes to characters based on their frequency. Characters that occur more frequently are assigned shorter codes, which leads to more efficient compression when the data contains frequent symbols.

3.2 Implementation in Python

Huffman encoding was applied to the output of the LZW compression, which serves as the source for calculating symbol frequencies. The Huffman tree was built using these frequencies, and the LZW-encoded output was further compressed by replacing the fixed-length LZW codes with variable-length Huffman codes.

The combined compression rate using LZW followed by Huffman encoding was 0.3999, meaning the folktale was compressed to 39.99% of its original size.

3.3 Efficiency of Huffman Encoding

Huffman encoding significantly improved the compression rate because the LZW-encoded output contained repeated patterns that were efficiently encoded by Huffman's variable-length codes. This improvement occurred due to a reduction in data redundancy and better utilization of symbol frequency.

4 Code Structure and Performance

4.1 Code Components

The following Python modules were used in the project:

- `lzw.py`: Contains the LZW compression and decompression logic, including dictionary management and string parsing.
- `huffman.py`: Implements the Huffman encoding and decoding processes, along with the construction of the Huffman tree based on symbol frequencies from the LZW output.
- `main.py`: Serves as the main script that ties together the LZW and Huffman functions, handling file input/output and triggering the compression/decompression processes.

4.2 Challenges and Solutions

One of the main challenges faced was managing large dictionaries in the LZW implementation, which caused memory usage to spike during compression. This was mitigated by ensuring that the dictionary was cleared once it reached a certain size. Ensuring the decompression process worked correctly also required careful testing, particularly with edge cases involving dictionary resets.

4.3 Performance Evaluation

The solution was tested on the medium-length folktale *Askeladden som stjal sølvendene til trollet*, and both speed and memory usage were found to be satisfactory. The use of dictionary resets in LZW and the efficiency of Huffman encoding made the program perform well for this size of text.

5 Results and Discussion

5.1 Compression Results

The following compression rates and performance metrics were achieved:

- **LZW compression rate:** 0.6578
- **LZW + Huffman compression rate:** 0.3999
- **Compression ratio:** 1.52:1
- **Compression time:** 0.0161 seconds
- **Decompression time:** 0.0035 seconds

Huffman encoding improved the compression rate by approximately 25.79%.

5.2 Trade-offs

The trade-off between compression time and efficiency was evident, particularly with Huffman encoding, which required more processing time than LZW. However, the reduction in file size justified the additional time complexity. In cases where speed is more important than compression efficiency, LZW alone might suffice.

5.3 Insights and Improvements

The compression results indicate that combining LZW and Huffman is highly effective. Future improvements could include experimenting with adaptive Huffman encoding or using dynamic dictionaries in LZW to achieve even better compression rates.

6 Conclusion

In conclusion, the implementation of LZW and Huffman encoding successfully compressed the folktale to a small fraction of its original size. The project provided valuable insights into the trade-offs between compression efficiency and computational complexity.