université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Introduction to the functional programming project report

AUTHORS

Yehor KOROTENKO - yehor.korotenko@etu-upsaclay.fr
Ivan KHARKOV - ivan.kharkov@etu-upsaclay.fr

2025-01-26

Contents

1 Introduction 1

2 Types of data used 2

3 Compression and Decompression Implementation 4

4 Testing Methodology 6

5 Compression Statistics 7

6 Debugging and Refinements Through Testing 9

7 Possible Improvements 10

8 Conclusion 10

1 Introduction

In this project, we implemented a Huffman compression algorithm to explore lossless data compression techniques and evaluate their efficiency across various types of text files. Huffman coding is a widely used algorithm that assigns variable-length binary codes to characters based on their frequency, allowing frequently occurring symbols to be represented with shorter codes and less frequent symbols with longer codes. This results in significant space savings for certain types of data, particularly those with non-uniform character distributions.

The goal of our project was to develop a functional and optimized Huffman compression system capable of encoding and decoding text files while maintaining lossless compression. We focused on key aspects such as building the Huffman tree, implementing an efficient min-heap for symbol prioritization, and ensuring correct file handling for compression and decompression. Additionally, we tested our implementation on various types of text data to analyze its strengths and limitations in different scenarios.

A structured development approach was adopted, beginning with the construction of a basic Huffman tree algorithm, followed by the addition of supporting modules such as file handling, unit testing, and debugging mechanisms. Git was used for version control to maintain a structured workflow and facilitate collaboration.

Given the complexity of Huffman compression, we divided the tasks strategically to ensure an efficient workflow and successful implementation.

Team members and responsibilities

- Yehor Korotenko:
  - Implemented Huffman Compression: Developed the core Huffman compression and decompression algorithms, optimizing them for performance and accuracy.
  - Huffman Tree Construction
  - File Handling & Decompression: Managed the processes for reading input files and decompressing data.
  - Testing: Collaborated on testing the overall compression and decompression functionalities, ensuring integrated system performance.
  - Interface Development
  - Data Handling Optimization
- Ivan Kharkov
  - Writing Compressed Files: Developed the functionality to output compressed data, ensuring adherence to Huffman encoding standards.

- Unit Testing: Designed and executed unit tests to validate individual components, guaranteeing their correctness and reliability.
- Heap Implementation: Implemented the min-heap data structure, a critical component for constructing the Huffman tree efficiently.
- Module Structure Development

Version Control and Branch Management

We utilized Git for version control, organizing our work through dedicated branches to maintain a structured development process and avoid conflicts. Initially, we developed a basic Huffman tree algorithm as a foundation for the project. Once this core functionality was established, we expanded it by integrating additional modules in separate branches. Each task was handled independently **-kharkov** for heap implementation, unit testing, and writing compressed files, and **korotenko** for Huffman compression, file handling, decompression, documentation, and debugging. This modular approach ensured that each component was implemented and tested in isolation before integration. Once a feature was finalized and verified, it was merged into the **main** branch, which contained the stable and fully integrated version of the project.

Workflow

Our workflow was structured around incremental development, starting with the core Huffman tree and progressively building surrounding components. Each team member worked within their respective branch, making regular commits to document progress and facilitate tracking. Code reviews were conducted frequently to ensure consistency, identify potential issues, and refine implementations where necessary. This helped maintain a clear and maintainable codebase while minimizing integration issues. Once a feature was complete and validated, it was merged into the main branch, followed by final testing to ensure system stability. Throughout development, debugging sessions were held to address unexpected behaviors and optimize performance. This methodical approach allowed for efficient collaboration while keeping the project structured, adaptable, and well-maintained.

With these essential processes in place, we proceeded to implement and optimize the core data structures that underpin the Huffman compression algorithm.

2 Types of data used

In the development of this project, several key data structures were employed to facilitate efficient implementation and optimization. The primary data structures used include Heap, Huffman Tree, Table of Occurrences, Table of Bytes, Table of Compressed Bytes, and Table of Conversion. These structures provided a robust base for the realization of the project.

Heap

A heap data structure was implemented to enhance the efficiency of constructing the Huffman tree. Specifically, a min-heap was chosen due to its ability to swiftly retrieve and remove nodes with the smallest frequencies - an essential operation in the iterative process of building the Huffman tree.

- `size: int`

  Represents the number of elements currently stored in the heap.

- `elements: ('a * 'b) list`

  Maintains the heap elements as a list of tuples, where each tuple typically consists of a character and its corresponding frequency.

The utilization of a min-heap ensured that the nodes with the lowest frequency were prioritized, thereby optimizing the tree construction process.

Huffman tree

The Huffman tree is playing a pivotal role in the encoding and decoding processes. This binary tree structure was useful in generating a table that associates unique binary codes with each character, replacing the conventional ASCII codes. The Huffman tree not only facilitated efficient data compression but also provided a mechanism for verifying the correctness of the transformations applied.

- `Leaf of int`

  Represents a leaf node containing the ASCII code of a character.

- `Node of tree * tree`

  Represents an internal node that combines two subtrees.

- `Nil`

  Empty tree, serving as the base case for recursive tree-building functions.

The nature of the Huffman tree ensured that more frequent characters were assigned shorter codes, thereby optimizing the overall compression ratio.

Table of Occurrences

The Table of Occurrences records how often each character appears in the text. It is needed to build the Huffman tree, as characters that appear more often get shorter codes. Stored as a list of tuples:

- First value: Character's ASCII code
- Second value: Frequency in the text
- Example: [(97, 10), (98, 5), (99, 2)] $\rightarrow$ 'a' appears 10 times, 'b' 5 times, 'c' 2 times.

Table of Conversion

The Table of Conversion maps each character to its Huffman code after building the tree. It is used for both encoding and decoding. Stored as a list of tuples:

- First value: Character's ASCII code
- Second value: Assigned Huffman code
- Example: [(97, '00'), (98, '01'), (99, '10')] $\rightarrow$ 'a' $\rightarrow$ '00', 'b' $\rightarrow$ '01', 'c' $\rightarrow$ '10'.

3 Compression and Decompression Implementation

Compression

The compression process begins by reading the input file byte by byte. For each byte encountered, the algorithm counts the number of occurrences of each unique byte, thereby generating a frequency table. This frequency table is then converted into a min-heap structure, which is utilized to construct a Huffman tree (huff_tree). The Huffman tree facilitates the generation of a Huffman coding scheme, where each unique byte is assigned a corresponding compressed binary representation. Specifically, the Huffman tree is traversed to create an array of pairs in the format [char (as integer), compressed code (e.g., 11110, 000001)].

The next critical step involves writing the frequency table to the compressed file. Initially, the length of the table is written to indicate the number of elements that need to be reconstructed during decompression. Following the table length, each character is written in its integer byte representation. Subsequently, the compressed bytes, represented as bit sequences, are written to the file.

To efficiently store the compressed data, four distinct templates for compressed bytes are utilized:

1. All Zeros: 00000000
2. All Ones: 11111111

3. Ones Ending with a Single Zero: 11111110
4. Zeros Ending with a Single One: 00000001

Templates 1 and 2 (all zeros and all ones) are unique and appear only once in the frequency table. In contrast, templates 3 and 4 can appear multiple times. During the reading process, sequences of identical bits are read until a change is detected, signaling the end of a compressed byte. For example, encountering the sequence 111110 indicates the end of a byte with the compressed code 11111.

However, a potential issue arises with templates 1 and 2, as their bit sequences do not include a bit change to indicate the end of a byte. To address this, these unique templates are handled separately at the beginning of the table. Specifically, after writing the table length and the uncompressed bytes, the algorithm writes the 11111110 byte and the 00000001 byte to represent the all ones and all zeros templates, respectively. This ensures that during decompression, these unique cases are correctly interpreted without ambiguity.

After writing the table, the file is reread to convert each byte into its corresponding compressed binary form, thereby minimizing the space required in RAM.

Decompression

The decompression process begins by reading the first byte of the compressed file, which indicates the length of the frequency table (tab_len). The next tab_len bytes are then read to reconstruct the uncompressed characters. Following the table reconstruction, the algorithm proceeds to read the compressed bit sequences.

Initially, the algorithm handles the unique cases of all ones (11111110) and all zeros (00000001). Upon encountering a bit change (from 1 to 0 or from 0 to 1), the algorithm identifies the end of a compressed byte. For the all ones case, the trailing 0 is removed to retrieve the original compressed byte 1111111. Similarly, for the all zeros case, the trailing 1 is removed to obtain the original compressed byte 0000000.

For subsequent bytes, the algorithm continues to read bits and identifies the end of each compressed byte upon detecting a bit change. Each complete compressed byte is then mapped to its corresponding uncompressed character using the previously reconstructed Huffman table. This process continues iteratively until all tab_len compressed bytes have been processed.

Finally, the algorithm reads the remaining bits in the compressed file bit by bit, accumulating them into a sequence. After each new bit is added, the algorithm checks whether the accumulated bit sequence matches any entry in the Huffman table. If a match is found, the corresponding uncompressed byte is written to the output file. This process repeats until the entire compressed file has been decompressed.

4 Testing Methodology

Unit testing

The testing process was conducted using *unit-test.ml*, executed with the command:

```
dune exec ./unit_test.exe
```

Testing Huffman Encoding and Decoding

- **Occurrence Calculation:** Ensured accurate frequency counts of each symbol.
- **Huffman Tree Construction:** Verified the correct formation of the Huffman tree based on symbol frequencies.
- **Print Tree Function:** Provided visual verification of the constructed Huffman tree structure.
- **Encoding Representation:** Checked the binary representations assigned to each symbol.
- **Compression:** Tested the ability to compress data correctly.
- **Decompression:** Ensured that decompression restores the original data.
- **Validation:** Confirmed that the decompressed data matches the original input.

Unit Tests for Various Strings

A comprehensive set of test cases was executed to validate the implementation, including:

- Simple repetitive sequences
- Simple characters and whitespace
- Sentences with various characters
- Empty strings
- Special characters
- Number sequences
- Unicode characters
- Large inputs and complex patterns

Additional Testing

- **Testing Min-Heap Implementation:** A series of assertions were performed to ensure the min-heap functions correctly, which is critical for building the Huffman tree.
- **File-Based Compression and Decompression:** The program was tested to write compressed data to files and decompress them back, verifying the correctness of file-based operations.

Results

The program outputs the Huffman tree structure, encoded bit streams, and decoded results. Final assertions confirm that decompressed data matches the original input, indicated by **Success** or **Error** at the end of each test.

Testing on Files

To evaluate the effectiveness of our Huffman coding implementation, we conducted a series of tests using various input files. These tests aimed to verify:

- Correct handling of edge cases
- Performance on Different File Types
- Compression Efficiency
- Lossless Compression

5 Compression Statistics

- ***book.txt***
  - **Description:** EBook
  - **Compression Statistics:**
    - **Original size:** 786,816 bytes
    - **Compressed size:** 714,646 bytes
    - **Space saved:** 9%
- ***book2.txt***
  - **Description:** EBook
  - **Compression Statistics:**

        - **Original size:** 338,268 bytes
        - **Compressed size:** 267,922 bytes
        - **Space saved:** 20%

- ***mid_long.txt***
    - **Description:** File containing mid-length text
    - **Compression Statistics:**
        - **Original size:** 19,635 bytes
        - **Compressed size:** 14,562 bytes
        - **Space saved:** 25%

- ***lot_of_same_letters.txt***
    - **Description:** File containing long sequences of two letters
    - **Compression Statistics:**
        - **Original size:** 57 bytes
        - **Compressed size:** 17 bytes
        - **Space saved:** 70%

- ***equal.txt***
    - **Description:** File created with a Python script containing repeated sequences of unique symbols
    - **Compression Statistics:**
        - **Original size:** 5,800,000 bytes
        - **Compressed size:** 11,575,177 bytes
        - **Space saved:** -99%

- ***equal.txt***
    - **Description:** File containing a couple of symbols
    - **Compression Statistics:**
        - **Original size:** 3 bytes
        - **Compressed size:** 7 bytes
        - **Space saved:** -133%

Interpretation of results

Efficient Compression For larger files such as *book.txt*, *book2.txt*, and *mid_long.txt*, the Huffman algorithm demonstrates moderate efficiency, achieving space savings between 9% and 25%. This indicates that the algorithm effectively reduces file sizes when dealing with substantial amounts of data. High Efficiency with Repetitive Symbols Files containing repetitive symbols, like *lot_of_same_letters.txt*, exhibit high compression efficiency, saving up to 70% of space. This showcases the Huffman algorithm's strength in handling data with dominant symbols, allowing significant space savings. Inefficiency with Unique Symbol Sequences However, files with repeated sequences of unique symbols, such as *equal.txt*, result in substantial space loss (-99%). This inefficiency arises because all symbols are equally repeated, leading to an average encoding length that surpasses standard encoding methods.

Mathematical explanation

The mean length per symbol can be calculated using the following approach: For an even number of unique symbols n:

$$\text{Mean length} \approx (\frac{n^2 + 6n}{4}) \cdot (\frac{1}{n})$$

Applying this to *equal.txt* where $n = 58$:

$$\text{Mean length} \approx (\frac{58^2 + 6 \cdot 58}{4}) \cdot (\frac{1}{58}) \approx 16$$

This is double the standard ASCII encoding of 8 bits per symbol, rendering the Huffman algorithm inefficient for files with a high number of unique symbols.

Inefficiency with Small Files

For very short files like *short.txt*, the algorithm incurs space loss (-133%). This is because the overhead of writing the Huffman table outweighs any potential compression benefits, making it unprofitable for small files. If the file is small, there are not enough repetitions to achieve significant compression.

6 Debugging and Refinements Through Testing

Initially, our implementation stored all data in a stack after reading, which functioned correctly on certain operating systems. However, testing on different systems with large files like *equal.txt* led to stack overflow errors. To address this, we transitioned to direct writing, which initially resolved the overflow issue. Nonetheless, we later discovered that the program was losing information during encoding and decoding, as stack overflow errors were no longer visible.

The final resolution involved replacing recursive functions with while loops, thereby preventing stack overflows and ensuring data integrity during compression and decompression processes.

7 Possible Improvements

One way to enhance the Huffman algorithm is by targeting larger patterns within the text instead of just individual characters. Currently, the algorithm compresses text by assigning shorter codes to more frequent single characters. However, many documents contain repeated sequences of characters or words. By identifying these common sequences and treating them as single units, the algorithm can encode them more efficiently. For example, if a particular phrase or a series of letters appears frequently, compressing the entire sequence at once can reduce the total number of bits needed, leading to better overall compression.

Combining Huffman coding with other compression techniques can also lead to better results. For instance, integrating Run-Length Encoding (RLE) or Lempel-Ziv (LZ) algorithms with Huffman coding allows the strengths of each method to complement one another. RLE is effective at compressing consecutive repeated characters by representing them as a single character followed by a count, while LZ algorithms excel at finding and encoding repeated substrings. By first applying RLE or LZ to simplify the text patterns and then using Huffman coding to efficiently encode the processed text, the overall compression ratio can be significantly improved compared to using Huffman coding alone.

8 Conclusion

At the end of this project, we have a program that:

- Compresses a .txt file into a .hf file using Huffman encoding.
- Decompresses a .hf file back into its original .txt format.
- Provides compression statistics, including file size reduction and efficiency.