|  |  |
| --- | --- |
|  | **PES UNIVERSITY**  **(Established under Karnataka Act No. 16 of 2013)**  **100 Ft. Road, BSK III Stage, Bengaluru – 560 085**  **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING** |

|  |  |  |
| --- | --- | --- |
| **Course Title: Problem Solving with C Laboratory** | | |
| **Course code: UE19CS152** | | |
| **Semester: II sem** | **Section: J** | **Team Id: 8** |
| **SRN: PES2UG19EC163** | **Name: VISMAYA IYER** | |
| **SRN: PES2UG19EC164** | **Name: YASHVITHA** | |
| **SRN: PES2UG19CS253** | **Name: NETRA SHALIGRAM** | |
| **SRN: PES2UG19CS372** | **Name: SERENA A. GOMEZ** | |

PROJECT REPORT

# Problem Statement

File compression involves the analysis of the symbols used in an input file, generation of a new encoding scheme for the symbols, and rewriting of the file using the new coding scheme. The number of bits used in the output file as a result of this new encoding scheme, is typically lesser than that in the input file.

In this project, we implement and evaluate the following encoding algorithm for text files:

* Huffman encoding
* LZW encoding & decoding

The scope of this project does not include the generation of the compressed files.

**Description**

The project is implemented using the following modules:

1. Zipper CLI
2. LZW encoder
3. LZW decoder
4. Huffman encoder

## Module: Zipper CLI

This module allows us to enter the file names as arguments (names of the file to be compressed, the compressed file and the file containing the code table) in the command line. It also allows us to run the compression and decompression of the file separately using the menu.

## Module: LZW Encoder

In this module we use the LZW encoding algorithm. The algorithm works by initializing the single characters scanning through the input string for successively longer substrings until it finds one that is not in the code table. When such a string is found, the index for the string without the last character (i.e., the longest substring that *is* in the code-table) is retrieved from the code-table and sent to output, and the new string (including the last character) is added to the code-table with the next available code. The last input character is then used as the next starting point to scan for substrings.

In this way, successively longer strings are registered in the code-table and available for subsequent encoding as single output values. The algorithm works best on data with repeated patterns, so the initial parts of a message see little compression. As the message grows, however, the compression ratio tends asymptotically to the maximum (i.e., the compression factor or ratio improves on an increasing curve, and not linearly, approaching a theoretical maximum inside a limited time period rather than over infinite time).

## Module: LZW Decoder

In this module, the decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized decode table. To rebuild the decode table in the same way as it was built during encoding, it also obtains the next value from the input and adds to the decode-table the concatenation of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value cannot be decoded (If the next value is unknown to the decoder, then it must be the value added to the decode table this iteration, and so its first character must be the same as the first character of the current string being sent to decoded output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the decode table.

In this way, the decoder builds a decode table that is identical to that used by the encoder, and uses it to decode subsequent input values. Thus, the full decode table does not need to be sent with the encoded data. Just the initial decode table that contains the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than explicitly sent with the encoded data.)

## Module: Huffman Encoder

In this module, the Huffman algorithm works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1 internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (excluding the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   1. Remove the two nodes of highest priority (lowest probability) from the queue
   2. Create a new internal node with these two nodes as and with probability equal to the sum of the two nodes' probabilities.
   3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

# C Concepts used

* File handling
* Structures
* Dynamic memory allocation
* Pointers
* Pre-processor directives

# Learning Outcome

* Implementation of Huffman encoding algorithm in C.
* Implementation of LZW encoding and decoding algorithm in C.
* Use of hash table

# Output Screenshots

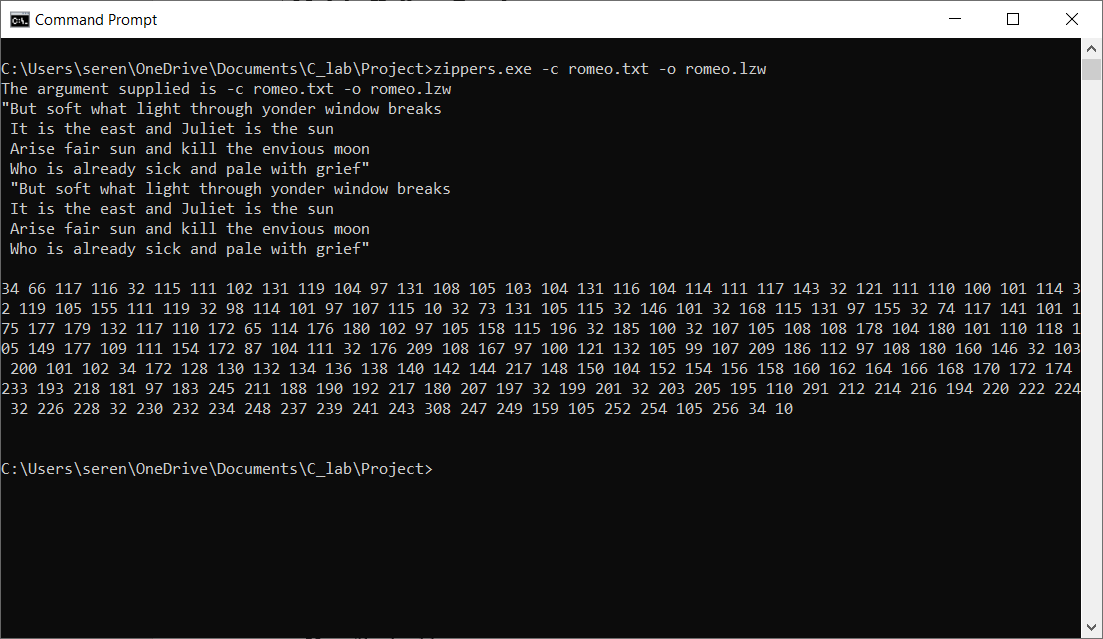


Figure 1 - LZW Encoding

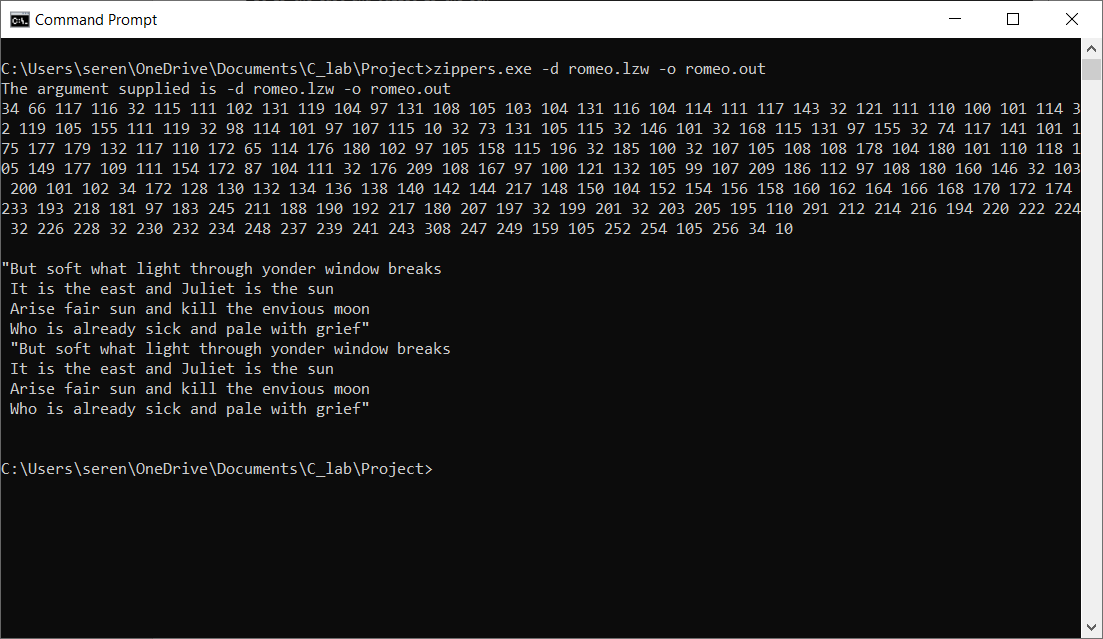


Figure 2 - LZW Decoding

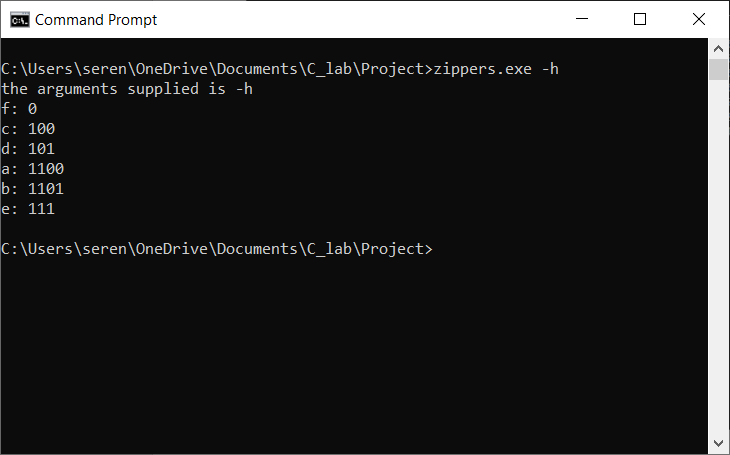


Figure 3 - Huffman Encoding

# Name and Signature of the Faculty