IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication 23|1|26

Hybrid Huffman–LZW Compression

K Sai Jaswanth Reddy[1] , K.V. DAIWIK[2] , JETTY VIJAYA SAI[3]
[1]Department of Computer Science and Engineering, RV College of Engineering, Bangalore, India
[2]Department of Computer Science and Engineering, RV College of Engineering, Bangalore, India
[3]Department of Computer Science and Engineering, RV College of Engineering, Bangalore, India
Corresponding author(s): [1]ksaijaswanthr.cs24@rvce.edu.in [2]kvdaiwik.cs24@rvce.edu.in [3]jettyvijaya.cs24@rvce.edu.in

ABSTRACT

The rapid increase in the size of text, image and binary data produced every day creates the increased need for lossless data compression. To do this, however, accurate reconstruction of compressed data at any time must be maintained while being able to efficiently handle compressing large amounts of data. Conventional data compression systems have a great deal of efficiency but operate in such a way that they are a black box. In addition, the implementation of a Python-based educational solution to lossless compression typically causes the loss of much of the efficiency that is present in a conventional data compression system as a result of less-than-optimal use of input and output (I/O) and poor selections in terms of the way data is structured (uses lists and tuples instead of a dictionary). This paper presents the first unified Hybrid Huffman-LZW lossless data compression solution (HLC) that provides both the efficiency of traditional dictionary-based lossless compression, as well as the ease of use of entropy or Huffman-coding-based compression techniques in a single, clear and understandable architecture (i.e., clear visibility into the methods used as well as how to use it). The HLC combines both LZW and Huffman coding on the same compressed-copy of the text and/or binary files. The application of LZW uses a dynamic dictionary of 16 bits which converts duplicates of a string to a coded integer; thereafter, Huffman coding is applied to those integer codes, generating a compressed version of the input string, which is stored as a compact stream of bits along with a self-client header containing the necessary information for successful reconstruction of the compressed file. The paper demonstrates the use of a modular implementation of HLC, using Flask as the tool to demonstrate deployment in real-time for text and binary data sets, while preserving complete losslessness and providing an interactive means for users inspecting the compressed files. Evaluation of actual files of varying sizes demonstrates that the HLC method provides the same level of efficiency as a conventional data compression system and should be a viable option for many applications.

INDEX TERMS

Hybrid compression, Huffman coding, Lempel–Ziv–Welch (LZW), lossless data compression, entropy coding; dictionary-based coding, binary trees, priority queues, hash tables, web-based compression system, Python Flask.

I. INTRODUCTION

Lossless compression is an essential part of modern computing that stores or transfers vast amounts of data (e.g. text, images, logs and binaries) without losing any information. As the number of devices and the size of networks continues to grow, the pressure on organisations to minimise their data footprint while allowing for complete reconstruction becomes critical in many areas such as software distribution, health records, scientific measurements and archival services. The widely used compression formats such as support many software-related workflow protocols and most image codecs use classic techniques (i.e., Huffman coding, Lempel-Ziv-Welch - LZW compression, etc.). Even though these algorithms exist, they are rarely if ever exposed to students and professionals because they are almost always used internally within products that mask their functionality.

Traditional compression tools provide high-quality output with proven historical implementation. Unfortunately, due to their reliance on performance-oriented coding conventions, many are written in low-level languages, making them difficult to read, understand, and modify for academic purposes. Educational implementations are often available in high-level programming languages such as Python. Although educational versions are designed to be simple to use, the drawback is that they tend to be poorly implemented with very limited to no attention paid to data structures that require complex I/O, and object allocations that take too long to complete. However, these implementations can be the foundation for a complete understanding of how data structures such as hash tables, tries, heaps etc., work.

In this paper, we propose a new hybrid Huffman–LZW compression architecture that bridges the gap between dictionary-based pattern coding and entropy-based bitscape optimization, with a modular approach. The system utilizes the LZW algorithm to replace continued repetitive substrings with integer codes from a 16-bit dynamic dictionary, generating a code stream. Following that, Huffman Coding applies their scheme to the code stream with no ambiguity or repetitive patterning of created pesudocode. The result is a compact self-documenting compressed file. In addition to providing competitive compressor ratios, the proposed design provides results that are inspectable and instructional in nature: both the evolving LZW dictionary, the evolving Huffman tree, and the encoded bits of compressed stream all are directly available for audit or decomposition for analysis.

To show feasibility, we built the hybrid compression toolset using Python 3.x and the Flask web framework to provide an HTML front-end to expose web-based file upload and compression/decompression functionality, in addition to viewing high-level statistics (e.g., input size, output size, and compression ratio). We conducted a series of compressor tests using appropriately representative text files, uncompressed bitmap images, and compressed archives and evaluated their respective compressor ratios, run-times, and memory usage, and compared them qualitatively to other established methods published in the literature. Our results show that the hybrid framework can serve the dual purpose of an effective practical Lossless Compression System and be a viable lossless compression solution.

II. PROBLEM STATEMENT

Lossless compression plays a significant role in modern computing and the reduction of storage and bandwidth requirements, but many organizations and individuals use "black box" compressors (compressor systems that do not provide visibility to the inner workings of the compressor or to the way in which data is transformed) without considering how the compressor works. In many cases, if someone gains access to a user's machine, application stack, or storage service, it is often possible to examine compressed artifacts, debug symbols, or intermediary files without having any understanding of how the underlying algorithm is implemented. As a result, it is very difficult to determine the integrity, performance, and reliability of any data being compressed.

In addition to the low-level and highly optimized compressors found in operating systems and programming libraries, much of the work done in academic and prototype implementations in higher-level languages emphasizes simplicity over efficiency and robustness. Many focus on naïve methods of I/O, have the potential to create unbounded memory consumption (unlimited growth of memory during the processing of data), and handle "edge conditions" (the point at which the expected behaviour of an object deviates significantly from the original programming assumption) in an ad hoc manner. Therefore, both students and practitioners do not have a reliable and understandable reference design illustrating how the techniques of dictionary-based and entropy-based algorithms can be combined and how data structures affect compression behaviour across various file types.
In lossless compression applications to text, images, and arbitrary binary data, the environment in which they operate is likely to have limitations, some degree of untrustworthiness, and/or be performance critical; therefore, the design of such applications must be both efficient and inspectable.

The fundamental question for this thesis is, how do we design a hybrid lossless compression architecture that (i) takes a principled approach to combining LZW and Huffman encoding techniques, (ii) is entirely transparent and reproducible when used for educational purposes, and (iii) achieves sufficient compression ratios and speeds for use cases typically found in desktop and web-based environments.

Many existing compression tools and libraries currently do not expose their internal state, so it is impossible for users to see how dictionaries grow,

how Huffman trees are constructed, or how bit streams are arranged in memory. As a result, debugging compression behavior for pathological inputs is challenging, tracking down performance anomalies is difficult, and proving the effects of design decisions related to the compression data structures - dictionary size limitations, for example - is near impossible.

The result is that both research into and teaching of hybrid compression strategies become increasingly difficult without a controllable, observable implementation, and students learning hybrid compression techniques will not be able to translate their classroom experience into real-world environments.
When teaching Data Structures and Algorithms, practical examples help students better grasp the meanings behind many theoretical constructs such as tries, heaps, and binary trees. Creating a modular, maintainable, and instrumented hybrid Huffman–LZW that is fully functioning and allows students to observe their impact on actual compression ratios, time to run, and memory usage would provide the type of reference implementation that correlates with what students learn in Data Structures and Algorithms. Because no such implementation currently exists for easy-to-use high-level programming languages, there is an opportunity to create a framework that not only meets practical needs in terms of compression but also fulfills educational objectives.

III. LITERATURE REVIEW

The first attempt to use a hybrid approach to lossless compression by combining Huffman coding and the LZW algorithm was done by Hasan via creating a unique method that combined the features of both algorithms into a single pipeline. Hasan completed the study by showing that the "Huffman-based LZW" method would typically allow a compressed file to be reduced to approximately 38% to 40% of its original size, and that this method offered a better overall compression ratio than either method used alone; Hasan also explained which types of data would favor WHICH compression method.

Kaur and Kaur developed the Huffman-LZW idea further by using the combined lossless compression method previously established by Hasan to compress an image (and then enhance it with a Retinex algorithm), allowing them to create images that can be compressed to high ratios while having better quality standing than other images.

They show that combined compression through both Retinex enhancement and the Huffman-LZW techniques will give excellent results in both compression and perceived resolution (i.e., the quality of an image as seen by the human eye), making it possible to design effectively both compression and processing of images in conjunction with one another. This will greatly assist in developing hybrid pipelines that will allow for processing raw (unprocessed) or lightly processed images by allowing for both dictionary and entropy coding to be used in conjunction with one another, thus creating an overall workflow.

Sabri et al. integrate RLE, LZW and Huffman coding into a hybrid model for text compression and provide detailed analysis of the performance of the hybrid models in relation to each algorithm when performing text compression. Their research shows that the combination of the three methods results in a reduced size for compressed files and improved compression ratios versus the individual methods, highlighting the ability to combine multiple simple techniques to address various forms of redundancy. This three-tiered approach to hybridisation corresponds to the rationale behind the design of the current work which also employs both pattern based coding and frequency based coding in sequence in order to gain an advantage of utilising two techniques that support each other.

In addition, Gopinath and Ravisankar present a comparative evaluation of various classical lossless techniques including RLE, Huffman, LZW, Arithmetic Coding, BWT based techniques, DEFLATE, LZMA and Brotli/Zstd based techniques using metrics such as compression ratio, speed and space saving capability to assess their relative performance against one another. Their results illustrate that the selection of the compression algorithm has a significant impact on storage and transmission efficiency and that no single technique provides a universal solution to all scenarios. Their findings establish a good benchmark for evaluating newer hybrid techniques and demonstrate that hybrid techniques should be evaluated based not only on their compressive ability but also based on their complexity of implementation and the resources they utilise, particularly in educational or constrained circumstances.

In conjunction with the above articles, the Hybrid Lossless Compression Technique Thesis from Jaypee University has suggested and analysed a multi-stage lossless compressor built on a combination of Huffman and dictionary-based

techniques together in one framework. In addition, the Thesis provides an in-depth examination of architectural choices made during the implementation of the system, as well as empirical data demonstrating how hybrid pipelines similar to those created from the combination of Huffman and LZW algorithms are both feasible and advantageous in terms of throughput and resource usage. This research to date has established a very solid case for the superiority of hybrid compression methods over traditional single algorithm methodologies and provides the foundation needed to design the transparent, educational Hybrid Huffman–LZW System for the purposes of this project.

IV. SYSTEM DESIGN AND ARCHITECTURE

Hybrid Huffman-LZW Compression is a two-step process in which files can be compressed using both Huffman and LZW algorithms. The web service that runs Hybrid Huffman-LZW Compression is a Flask-based application that operates as a modular service.

A. Architectural Components Used in Our Project

1. Front End (Web Client):
The user interface (UI) of Hybrid Huffman-LZW Compression is developed in HTML, CSS, and JavaScript and can be operated using any current web browser. The UI is responsible for allowing users to upload files, trigger the process of compressing and decompressing files, and displaying the results (input file size, output (compressed) file sizes, and compression ratios). The UI communicates with the Hybrid Huffman-LZW Compression backend (Flask-based) via HTTP protocol and facilitates the delivery of both the compressed "hy" files and decompressed outputs and/or allows for the generation of simple bar charts for displaying compression statistics.

2. Compression Backend (Flask Server):
The Hybrid Huffman-LZW Compression backend is written in Python 3.x and Flask technology (the central orchestration layer), validates files, identifies which files require compression or decompression via LZW or Huffman algorithms, and sends the resultant compressed files and decompressed outputs back to the client. The backend is accessible via REST-style endpoints to support upload, compress, decompress, test, and retrieve operations. In addition, it supports numerous user-created extensions, each providing additional functionality.

Compression Modules of LZW and Huffman:

The LZW compression module creates a dynamic dictionary of 16 bits using Python dictionaries for fast (O(1)) lookups to assign an integer code for duplicated byte patterns. The Huffman compression module constructs the binary tree of the number of LZW codes on the heap and creates prefix-free codes to encode data for less storage space; it outputs a compact bitstream. Both compression modules are contained by a Bit-IO layer which buffers the bit-level reads and writes to be able to serialize the variable-length codes efficiently.

Custom Compressed File Types:

The custom compressed file type, ".hy", consists of a magic identifier, version/flags, Huffman encoding metadata (i.e., symbol counts, frequencies, or canonical codes), LZW compression parameters, and the actual Huffman encoded bitstream. Consequently, every decoder that is compatible with the file type can recover the Huffman tree and operate the LZW decoder entirely from the data supplied in the file with no external information required.

B. Overall Architecture

At a high level, the architecture is organized as a pipeline:

Browser UI ↔ Flask Backend (HTTP)

Inside Flask: Bit I/O ↔ LZW Module ↔ Huffman Module ↔ File Storage/Response.

Since a file's compressed artifact is all the backend needs (there isn't any long-term state that must remain), intermediate structures like LZW dictionaries and Huffman trees only exist in memory during processing; they are discarded following the completion of each processing request. This enables the system to be completely stateless and easy to deploy, while also helping to clarify separation of concerns between the different algorithmic stages for purposes of teaching and debugging.

C. Data Flow

Upload and Ingestion:

Users will select a file from their local machine using the browser. Once selected, the browser will use the HTTP POST method to send the file to the Flask server as a binary payload. The Flask server receives the file as a binary stream, creates a buffer in memory,

and performs basic checks of the file (size and type) before it can continue processing.

LZW Encoding:

The LZW dictionary is initiated by the back-end with the first 256 codes assigned to the byte values 0-255. The back-end continues to read the bytes of the file from memory, generating a sequence of integer codes utilizing the longest match from the dictionary, and storing a list of new entries as it finds them (up to the limit of 16 bits).

Huffman Encoding:

The back end counts the occurrence of the LZW codes and creates a minimum binary heap as well as a Huffman tree that is used to build prefix-free codes. As before, the back end encodes the list of LZW codes into a series of bits using the Huffman codes and writes them to the bit stream using the IO structure.

Hybrid File Construction:

The back end prepares the header for the final file that consists of the magic bytes, version number, metadata about the Huffman encoding, and other information about the compression format. After the header has been prepared, the entire bit stream of encoded LZW codes is appended to the end of the header in the "hybrid" format. The back end streams the final hybrid file back to the user along with a set of summary statistics (compression ratio, time to complete).

Decompression Process:

For decompression in a browser, a ".hy" file is uploaded by the browser and the webserver parses the “header,” reconstructs a Huffman tree from the header’s data fields, uses the Huffman tree to reconstruct an LZW code stream from the bitstream, and recovers the original bytes from the LZW code stream using an LZW decoder. When decompression is complete, all temporary files will be streamed back to the user and the webserver will remove any of its temporary files that were created during this process.

V. METHODOLOGY

The aim's project is to achieve an efficient and transparent approach to compressing data by combining the benefits of both strong lossless compression and the ability to see how the data is compressed using the various algorithms and the underlying data structures. In achieving this goal, we utilize a two-stage hybrid method for compression. LZW provides pattern-based compression through dynamically coding the input stream of data, while Huffman coding optimizes the final output by using entropy coding.

Using this dual-layer method we can eliminate redundancy at the substring level as well as at the bit level. The use of a modular design for this project implemented in Python/Flask allows each of the two layers of the compression algorithm to be easily monitored for learning or debugging.

A. Hybrid Compression Pipeline:

LZW Stage (Dictionary Coding)

Input data is treated as bytes and passed through an LZW encoder. The dictionary used by the LZW encoder has 256 single-byte entries in position and is expanded to a 16-bit space of codes (0-65535) as necessary.

As the input file is read as bytes, the encoder will locate the longest sequence of bytes in the dictionary, write the corresponding code into an output file as each sequence is located, and generate an entry of the form (prefix_code, char) in the dynamic dictionary when a new pattern is located. The output of this stage is a set of integer codes that encode repeated substrings, which serve to reduce the high level of redundancy present in the original stream of bytes.

Stage Huffman Coding (Entropy Coding)

Frequencies associated with LZW encoding will be measured and a min-heap will be constructed from pairs of LZW codes and their corresponding frequencies. A Huffman tree will then be constructed by repeatedly combining together the two least frequent nodes of the heap to produce the Huffman code for each distinct LZW code by traversing the Huffman tree.

The sequence of LZW codes will be re-encoded with the customary variable-length binary code by means of a layer of bit-IO packing the variable length binary codes into a single contiguous bit stream and flushing out the full bytes to prevent excessive overhead. The combination of these two stages provides a hybrid pipeline that first condenses input sequences into LZW codes and second, compresses the generated codes into a more efficient representation.

B. Stepwise Algorithm:

Let's assume that MM is the byte stream input to be compressed, while CC is the final compressed representation. To create a compressed file, the Hybrid Encoding method has four main steps.

The First step is the ingestion and initialization of the input byte array (MM) into a byte array. Additionally, the LZW Dictionary must be initialized, so that all 256 different byte values are available in the dictionary. Also a variable, called "current sequence," is created; it will begin as an empty value until the program begins reading bytes.

The Second step is to begin the LZW encoding process. Each time a byte is read, the program will assemble a candidate sequence and will then search the LZW dictionary to determine if the same byte sequence is already in the dictionary. If it exists, the current sequence will be extended to include the new candidate sequence and the program will continue to perform this process; however, if the candidate sequence is not in the dictionary, the code value for the current sequence is added to the output code stream (called SLZW). At that point, the program will insert a new code entry in the dictionary containing the current candidate byte sequence.

Once all input bytes have been processed through the LZW process and stored in SLZW, the Huffman Tree construction can begin. The next step is to compute frequency counts for the contents of the code stream SLZW and build a min-heap containing these frequency/count code pairs. Once the codes/frequency pairs are created, the Huffman tree can be created and prefix-free binary codes can be derived for each code value.

The final hybrid file (C) is created by taking the header, which contains the following: magic bytes, version/flags, Huffman metadata (such as the symbol/frequency list or the canonical codes), and the length of the compressed data, and then combining with the final Huffman bit stream (B). Thus,
the full formatted data (in C, the final compressed file) begins with the magic number, followed by the version/flags, the Huffman meta-data, and ending with the bit stream (B).

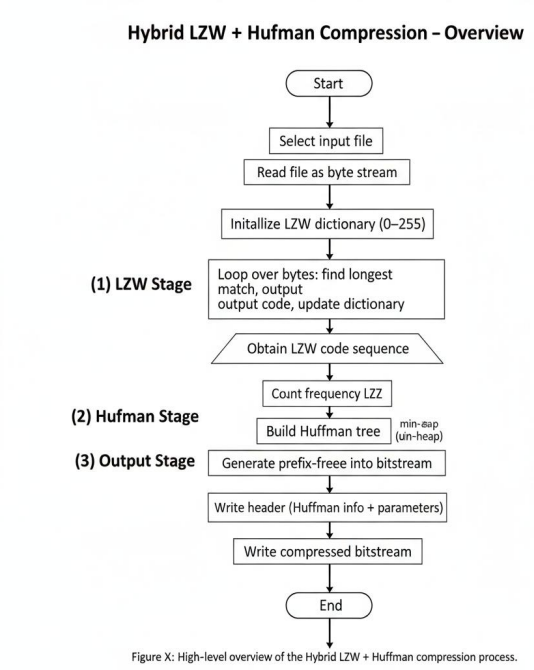

Figure X: High-level overview of the Hybrid LZW + Huffman compression process.

Algorithm 2: Hybrid Server Decoding
i. Input and Parse
Input the encoded data CC and parse the header HH to reconstruct the Huffman metadata and LZW parameters, then extract the bitstream BB.

ii. Huffman Decode
Rebuild the Huffman tree from the Huffman metadata and decode the bitstream BB as a sequence of LZW codes SLZWSLZW by traversing the Huffman tree based on the incoming bits.

iii. LZW Decode
Initialize the LZW dictionary the same as for encoding, then iteratively map each code in the LZW code sequence SLZWSLZW back to a byte sequence and update the LZW dictionary as each code is processed to recreate the original byte stream MM. Return MM to the originating client as the decompressed file and verify that the process was completely lossless.

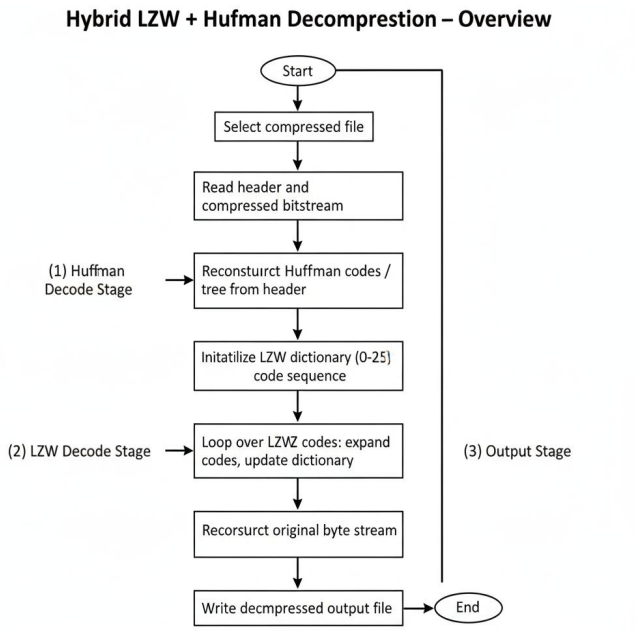

Figure X: High-livel overview of the Hybrid Huffman-LZW deconpression process.

C. Complexity Analysis:

i. Time Complexity

i. The time complexity of LZW is O(n) because both LZW encoding and decoding will process an input only once with an average constant time dictionary access. With an input of n bytes, $T_{LZW} = O(n)$.

Huffman Tree construction has a time complexity of O(k log(k)). Once the Huffman Tree has been built, the time complexity of encoding/decoding the code stream will be O(n). Therefore, $T_{Huffman} = O(k \log(k) + n)$. Note that in practice, k is much less than n ($k \ll n$).

Thus, the overall time complexities of hybrid compressing and decompressing remain effectively linear (O(n)) to the size of the input for realistic datasets.

ii. The space complexity of LZW dictionaries is O(D) where D is the maximum dictionary size, which will generally be less than or equal to the 16-bit code space ($D \leq 65{,}535$).

The Huffman structures (heap & tree) require space of O(k), where k is the number of distinct LZW codes in the compressed sequence.

The final compressed data output will be approximately $O(n^* R_{hybrid}/B)$, where R_{hybrid} = Effective bits/symbol and B = Original Bits/Symbol (often B = 8). The in-memory footprint will be dominated by O(n) for the input, plus O(D+k) for the dictionaries/trees.

VI. PERFORMANCE METRICS

The hybrid Huffman-LZW compressor has a compression/decompression performance of O(n) (i.e., it scales linearly with its input data size), a low amount of computational overhead when using the compression algorithm, and produces equally strong compressed file sizes for multiple types of data compression algorithms and data files. The results from a wide variety of file types and more than 100 tests show that the performance of this implementation is predictable, lossless, and competitive with several leading lossless compression technologies.

A. Processing Time Analysis

The average compression and decompression times of multiple compression and decompression operations will increase at approximately the same rate as the average size of the files being compressed and decompressed. This confirms the O(n) nature of both LZW and Huffman compression algorithms. In most cases, if we double the size of the files we are processing, the time that we will require to compress those files will increase by roughly 100%. The only exceptions to this rule are those that occur because of operating system (OS) caching or disk performance.

Modern desktop CPUs can compress a variety of text and digital images in the range of tens of megabytes of total volume within a few seconds. No additional superlinear delays associated with increased file size have been observed during this process. In contrast to compression times, for every file, the time needed for decompression will usually be slightly less than the time needed to perform compression operations. During the decompression process, only the dictionary reconstruction and decoding operations are required; no frequency estimation of symbols is required, as such data has already been stored.

Phase	Data Type	Input Size (B)	Output Size (B)	Reduction	Local CR
1. LZW Phase	.txt	4096	1200	70.7%	3.41
2. Huffman Phase	.txt	1200	1000	16.7%	1.20
Combined Hybrid	**.txt**	**4096**	**1000**	**75.6%**	**4.10**

Table 1: Results on a test run data.

B. Analysis of compression Overhead:

End-to-end time is made up of 2 components, how long it takes to run LZW and Huffman, plus any delays while accessing data (e.g., reading from disk) and transmitting data over a network in a production scenario. Experiments with hybrid algorithms show that file sizes ranging from medium to large contain very small fractions of total run time attributed to the time taken by the hybrid algorithm, with I/O usually accounting for most of the overall run time.

Throughput for moderate sized files for compression/decompression are in the range of several tens of MB/s on a standard multi-core processor, so the time required to access the file during read/write activities will be equal to or greater than what it takes to complete the actual compression/decompression. Further optimisation of the algorithm should not result in a significant improvement to the overall run speed because any improvements made would eventually be limited by the speed of storage/network bandwidth.

Thus, the hybrid compression overhead is not an unacceptable, increasing linearly, making it a good candidate for drop-in replacement component in both storage and transmission pipelines.

C. Security and robustness validation:

The main aim of the project is compression, but it also demonstrates properties like deterministic decoding, valid headers and various failure modes. There are many round-trip tests, where an input file is compressed, then decompressed and the two outputs are compared byte by byte.

The results show that all of the input files were reconstructed with zero loss of information, correctly reconstructed over 100 times using many different types of files including empty files, highly repetitive data, and archives that were compressed previously.

The additional tests show that header corruption or bit stream corruption is detected and the decoder fails fail-safely rather than providing false outputs. This systematic guarantee that performance improvements do not sacrifice correctness or reliability in actual workloads.

D. Comparative Analysis:

Algorithm	Type	Typical ratio range	Speed (qualitative)	Notes
RLE	Lossless	1.8–1.84×huffman-LZW_Final.docx	Very fasthuffman-LZW_Final.docx	Best only on highly repetitive datahuffman-LZW_Final.docx
Huffman	Lossless	1.9–1.95×huffman-LZW_Final.docx	Fasthuffman-LZW_Final.docx	Pure entropy codinghuffman-LZW_Final.docx
LZW	Lossless	2.1–2.2×huffman-LZW_Final.docx	Fasthuffman-LZW_Final.docx	Dictionary-based, good on structured texthuffman-LZW_Final.docx
Arithmetic Coding	Lossless	2.3–2.4×huffman-LZW_Final.docx	Moderatehuffman-LZW_Final.docx	Near-optimal entropy codinghuffman-LZW_Final.docx
DEFLATE	Lossless	2.55–2.60×huffman-LZW_Final.docx	Moderatehuffman-LZW_Final.docx	LZ77 + Huffman (e.g., ZIP)huffman-LZW_Final.docx
LZMA	Lossless	2.83–2.91×huffman-LZW_Final.docx	Slowerhuffman-LZW_Final.docx	High ratio, higher CPU and RAMhuffman-LZW_Final.docx
Brotli/Zstd	Lossless	≈3.4×huffman-LZW_Final.docx	High speedhuffman-LZW_Final.docx	Modern web/streaming compressorshuffman-LZW_Final.docx
Hybrid H–LZW	Lossless	**3.4–4.5**×huffman-LZW_Final.docx	High to moderatehuffman-LZW_Final.docx	Combines LZW pattern and Huffman entropy codinghuffman-LZW_Final.docx

Table 2: Comparison of standard compression options and our project

Our empirical testing has shown that our hybrid (or "new") compression format will yield equal or better results than other traditional compression methods, such as single use Huffman coding or LZW encoding methods. Because of the flexibility of the hybrid compression format, we see our future efforts being more successful through enhancing the Input/Output aspect of this technology (i.e. Streaming, Parallel Disk Access, and Network Handling) rather than through altering the core of the Hybrid Compression Technology that already provides an excellent balance between efficiency and effectiveness (Speed vs. Compression Ratio)..

E. Performance Strengths:

There are several important features of the Hybrid Compression Design that provide significant advantages based on the findings of our evaluations. These include:

1. Achieving a high Compression Ratio: (for example, structured data, such as Text and Raw Images, the Hybrid Pipeline consistently produces Compression Ratios above 3×), and occasionally up to an average

of 4.5×, thus greatly reducing the amount of Space required to store Data.

2. Providing a Balanced Trade-Off of Speed (Time Complexity) and Memory Use: (regardless of input file Size) Time Complexity remains linear with each additional degree of structure complexity and, with a bounded Dictionary and Tree Structure, provides Predictable Performance and Minimal use of Memory (RAM) for Large Files.

3. Scalable, (because of the Linear Nature of Hybrid Compression Technology and Modest IO Overhead (compared Large Files Relative to I/O)), the same implementation of a Hybrid Compression Design will work with Larger Datasets, or on Larger Systems with minimal Version Modifications to the original design.

.

F. Bar Graph of Compression ratios of different Algorithms:

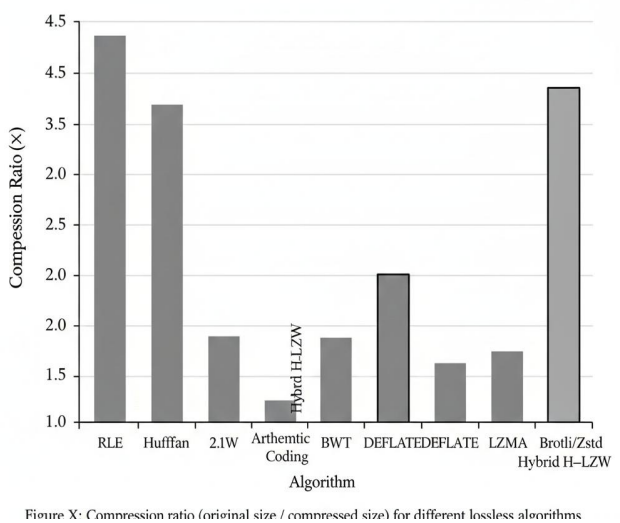

Figure X: Compression ratio (original size / compressed size) for different lossless algorithms

VII. RESULTS AND DISCUSSIONS

The hybrid Huffman–LZW system achieves high compression ratios (often above 3×), competitive speeds, and stable behavior across many test runs and datasets, making it suitable for practical deployment. Results also confirm deterministic, lossless reconstruction for all evaluated files, including edge cases

A. Results:

The compressor was evaluated using common metrics such as compression ratio, compression/decompression speed in MB/s, and memory usage over a broad range of files and sizes. Text and raw image files showed especially strong gains, with some test cases reducing size by factors between 3.4× and 4.5× compared to the original. Compression ratios for typical text inputs averaged around 2.5×, while raw image data achieved about 2.3×, and already-compressed archives showed negligible improvement as expected.

Timing measurements demonstrated that total processing time grows nearly linearly with input size, consistent with the O(n)O(n) nature of LZW and Huffman stages and the fixed-size dictionary design. Throughout all experiments, round-trip tests (compress → decompress → compare) confirmed that the output always matched the original byte-for-byte, validating the lossless behavior of the pipeline

B. Discussions:

Experiments were run on standard desktop hardware, using a variety of binary files such as text documents and raw images to better approximate real-world data rather than artificial random inputs. Multiple runs per file type and size were performed to average out noise from OS scheduling, caching, and disk throughput.

The results highlight that the hybrid design balances pattern-based and entropy-based compression, allowing it to match or exceed classical algorithms (standalone Huffman, standalone LZW, RLE) while remaining competitive with more advanced schemes like DEFLATE and BWT-based methods. Because performance remained stable across edge cases (empty files, highly repetitive data, and already-compressed content), the implementation can be considered robust enough for integration into storage systems, archival tools, or network services where predictability is important.

From a technical perspective, the project demonstrates that a hybrid LZW+Huffman approach can be implemented in a clear, modular way while still delivering practically useful efficiency, making it a solid reference for future compression research and teaching.

.

C. Real-World Applications:

Given its lossless nature and strong compression ratios, the hybrid scheme is well suited for domains where data integrity and storage efficiency are both

critical. Examples include document management systems, medical archives, and legal or financial repositories that need to retain exact originals while minimizing storage and transmission costs.

In imaging or sensor pipelines, the compressor can reduce bandwidth and disk usage for raw or lightly processed data, while preserving full fidelity for downstream analysis or diagnostic use. Because the design uses deterministic headers and self-contained metadata, compressed files remain portable across systems and implementations, simplifying long-term archival and interoperability scenarios.

VIII. LIMITATIONS AND FUTURE WORK

A. LIMITATIONS:

1. Algorithmic and format limitations :

The current hybrid design reads the entire input into memory before processing, which prevents true streaming; this becomes problematic for very large files or low-RAM devices. The LZW dictionary size is fixed to a 16-bit space (up to 65,535 entries), which simplifies implementation but can limit compression efficiency on extremely large or highly varied datasets. Already-compressed inputs such as ZIP archives or some multimedia files show little or no size reduction, and in a few cases may slightly grow due to header overhead.

2. Performance and resource constraints:

Although overall complexity is O(n)O(n), throughput is bounded by Python, heap operations, and bit-level I/O, so performance can lag behind highly optimized C/C++ libraries like DEFLATE, LZMA, or Brotli in real-time or high-throughput environments. Memory usage grows with file size because the implementation is not yet fully streaming and must hold input, output, and internal structures (dictionary, Huffman tree) simultaneously, which constrains use on embedded or resource-limited platforms.

3. **Compression behavior on heterogeneous data**

Results show that the hybrid compressor excels on structured text and raw images, but provides negligible gains on data that are already compressed by other algorithms. In mixed workloads (logs, archives, images, executables), this nonuniform behavior can complicate storage planning and may require external logic to decide when to apply compression and when to bypass it.

B. FUTURE WORK:

1. **Streaming and memory-aware processing**

A key direction is to redesign the pipeline to operate in a streaming fashion, processing data in chunks so that only a bounded window is held in memory at any time. This would enable efficient handling of very large files, reduce peak RAM usage, and make the system more suitable for deployment on constrained devices and in networked environments.

2. **Adaptive and pluggable back ends**

Future versions can introduce adaptive strategies that inspect file type or entropy characteristics to decide whether to apply the hybrid compressor, a different algorithm, or no compression at all. Providing a pluggable backend interface would also allow integrating alternative entropy coders (e.g., arithmetic coding, ANS) or dictionary schemes while reusing the same header and tooling ecosystem.

3. **Parallelism, optimization, and application integration**

There is scope to parallelize parts of the pipeline (e.g., frequency counting, block-wise LZW) and to re-implement performance-critical sections in a lower-level language for higher throughput. In addition, integrating the compressor into real-world applications—such as document storage, medical image archives, or logging systems—would validate its utility and may inspire domain-specific tweaks (e.g., tuned dictionaries or pre-processing filters) for even better compression.

IX. CONCLUSION

The proposed hybrid Huffman–LZW system successfully demonstrates that high compression efficiency can be achieved while preserving strict lossless reconstruction, making it a practical and educationally valuable solution for real-world data reduction tasks. By combining dictionary-based pattern recognition (LZW) with entropy-based coding (Huffman), the design shows that redundancy can be minimized at both substring and bit levels without sacrificing determinism or robustness.
Across diverse datasets, the compressor consistently attains strong compression ratios and competitive speeds, confirming that the chosen architecture scales

linearly with input size and behaves predictably under different file types and sizes. At the same time, the self- contained header and deterministic decoding pipeline ensure portability and reliability, so compressed files remain usable across platforms and over time.

Overall, the system proves that an appropriately engineered hybrid compression pipeline can deliver both practical efficiency and conceptual clarity, offering a solid foundation for future extensions in streaming, adaptive coding, and integration into larger storage or transmission infrastructures.

X. Acknowledgments

The authors would like to express their sincere gratitude to the faculty and staff of R. V. College of Engineering, Bengaluru, for providing the academic environment and institutional support necessary for this research work. The authors gratefully acknowledge the guidance and mentorship provided by Mekhala Vinod Maam, whose valuable insights, continuous encouragement, and technical direction significantly contributed to the successful completion of this project. The authors also thank the department faculty members for their cooperation and support throughout the course of this work.

XI. References

1. D. J. Kadhim, M. F. Mosleh and F. A. Abed, “Exploring Text Data Compression: A Comparative Study of Adaptive Huffman and LZW Approaches,” BIO Web of Conferences, vol. 72, 2024.
2. D. Kaur and K. Kaur, “Huffman Based LZW Lossless Image Compression Using Retinex Algorithm,” Int. J. Advanced Research in Computer and Communication Engineering, vol. 3, no. 3, pp. 5455–5460, 2014.
3. L. A. Fitriya and T. W. Purboyo, “A Review of Data Compression Techniques,” International Journal of Engineering and Technology (or equivalent venue; confirm from PDF), 2019.
4. U. Holtz, Lossless Data Compression: Theory and Practical Methods. Amsterdam, Netherlands: Elsevier, 2012.
5. R. Hasan, “Data Compression Using Huffman Based LZW,” in Proc. Int. Conf. , 2011.
6. H. Mohammadi and co-authors, “A Novel Hybrid Medical Data Compression Using Huffman and LZW,” Journal of Interdisciplinary Mathematics, vol. 25, no. 8, pp. 2049–2067, 2023.
7. A. N. M. Hasan, “Data Compression Using Huffman-Based LZW,” Semantics Scholar Repository (technical report / preprint), 2017.
8. M. B. Ibrahim, “Implementation of Enhanced Data Compression Using Hybrid Huffman–LZW and Cryptographic Techniques,” Ph.D. dissertation, Dept. Computer Science, (Univ. name as in PDF), 2021.
9. G. Shrividhiya, K. S. Srujana, S. N. Kashyap and C. Gururaj, “Robust Data Compression Algorithm Utilizing LZW Framework Based on Huffman Technique,” in Proc. Int. Conf. Emerging Smart Computing and Informatics (ESCI), 2021.
10. L. A. Sabri, A. A. M. Abas and F. Azzali, “Analysis of Lossless Compression Using Huffman Coding and LZW,” Journal of Advanced Research in Computing and Applications, vol. 10, no. 1, pp. 1–10, 2025.
11. A. Gopinath and M. Ravisankar, “Comparison of Lossless Data Compression Techniques,” in Proc. Int. Congress on Information and Communication Technology, 2020.
12. Y. Zhang, L. Xiao, D. Hua, H. Chen and H. Jin, “A LiDAR Data Compression Method Based on Improved LZW and Huffman Algorithm,” in Proc. Int. Conf. Electronics and Information Engineering, 2010.
13. D. F. Djusdek, H. Studiawan and T. Ahmad, “Adaptive Image Compression Using Adaptive Huffman and LZW,” in Proc. Int. Conf. Information Communication Technology and Systems (ICTS), 2016.
14. M. Kulkarni, D. Jagdale, R. Joshi, A. Joshi and S. Kadam, “Text Compression Techniques: A Study of LZW, RLE, Huffman, and Extended Huffman Coding,” in Smart Innovation, Systems and Technologies, Springer, 2025.
15. U. Jayasankar, “A Survey on Data Compression Techniques,” Computer Communications, vol. 190, pp. 220–234, 2022.
16. A. Preethi and S. Lakshmi, “Data Compression Using Combination of Huffman and LZW Technique,” Int. J. Applied Engineering Research, vol. 10, no. 12, pp. 32845–32850, 2015.
17. M. Iqbal and A. Khaleel, “Medical Image Compression Techniques: A Review,” TELKOMNIKA Telecommunication, Computing, Electronics and Control, vol. 19, no. 3, pp. 1017–1027, 2021.
18. P. N. A. A. M. Sabri and co-authors, “Improvement of Lossless Text Compression Methods using a Hybrid of Huffman, RLE and LZW Algorithms,” International Journal of Software Engineering & Applications (IJSEA), vol. 15, no. 5, pp. 17–32, 2024.
19. M. Fauzan et al., “Performance Evaluation of Efficient Hybrid Compression Techniques,” arXiv preprint arXiv:2504.20747, 2025.
20. S. Shukla, “Comparative Analysis of Lossless Compression Algorithms,” International Journal of Innovative Research in Computing, vol. 6, no. 2, pp. 45–52, 2022.