Exploring Data Compression Algorithms: Techniques and Implementations

MUHAMMAD KHUBAIB, Habib University, Pakistan
NEIL KANTH LAKHANI, Habib University, Pakistan
KUSHAL CHANDANI, Habib University, Pakistan

ACM Reference Format:
Muhammad Khubaib, Neil Kanth Lakhani, and Kushal Chandani. 2024. Exploring Data Compression Algorithms: Techniques and Implementations. 1, 1 (May 2024), 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Data compression is a vital process in the realm of computer science, enabling the efficient storage and transmission of digital information by reducing its size. In this paper, we embark on a comprehensive exploration, implementation, and analysis of various data compression algorithms. Our focus lies in understanding the mechanisms, efficiencies, and potential applications of these algorithms. Data compression techniques are broadly categorized into lossless and lossy compression. Lossless compression ensures that the original data can be perfectly reconstructed from the compressed data, whereas lossy compression allows for some loss of information to achieve higher compression ratios.

We will delve into innovative techniques such as:

(1) Huffman Coding
(2) Lempel-Ziv-Welch (LZW) Compression
(3) Run-Length Encoding (RLE)
(4) Burrows-Wheeler Transform (BWT)

Each of these algorithms offers advanced strategies for compression and has unique characteristics that make them suitable for different types of data. Our project aims to evaluate these algorithms across various metrics, including compression ratio, computational efficiency, and applicability to different types of data, especially text. We will also explore the trade-offs between compression and decompression speeds, memory requirements, and the potential for parallelization.

Through this comprehensive study, we aim to gain a deeper insight into the field of data compression, contributing to the efficient management of digital data.

Authors' Contact Information: Muhammad Khubaib, Habib University, Karacho, Texas, Pakistan, mk07218@st.habib.edu.pk; Neil Kanth Lakhani, Habib University, Karachi, Pakistan, nl07197@st.habib.edu.pk; Kushal Chandani, Habib University, Karachi, Pakistan, kc07535@st.habib.edu.pk.

Manuscript submitted to ACM

2 TECHNIQUES

2.1 Huffman Coding (Greedy Technique)

2.1.1 *Introduction*

. Huffman coding is a type of greedy algorithm used in lossless data compression. It prioritizes the most frequent characters with shorter codes and less frequent characters with longer codes, optimizing the overall space used. By building a binary tree from the bottom up, where each leaf node represents a character from the data set and each internal node holds the combined frequency of its children, Huffman coding effectively minimizes the average code length needed to represent each character, thus achieving efficient compression. This method is particularly useful when dealing with varied frequency distributions of characters within a data set.

2.1.2 *Design Approach.*

- **Input Data:** The process begins by determining the frequency of each character or symbol in the input data. For example, in a text file, each character's occurrence is counted.
- **Priority Queue Initialization:** Each character or symbol is treated as a node and added to a priority queue (often implemented as a binary min-heap), where nodes are prioritized based on their frequency—the lower the frequency, the higher the priority.
- **Tree Construction:** Huffman coding builds a binary tree bottom-up:
  - Extract the two nodes with the lowest frequency from the priority queue.
  - Combine these nodes into a new node, where the combined frequency is the sum of the two individual frequencies.
  - Insert the new node back into the priority queue.
  - Repeat the process until there is only one node left in the queue, which becomes the root of the Huffman tree.
- **Assigning Codes:** Starting from the root of the tree, traverse down to each leaf node:
  - Assign '0' for left branches and '1' for right branches.
  - The path from the root to each leaf node forms the binary code for the character at that leaf, ensuring that no code is a prefix of any other (prefix-free property).
- **Encoding the Data:** Using the generated Huffman codes, the original input data is then encoded into a compressed binary sequence, substituting each character with its corresponding Huffman code.
- **Decoding:** The data can be decoded by traversing the Huffman tree from the root to the leaves, following the bits of the encoded data, where each leaf represents a character.

2.1.3 *Theoretical Analysis*

. **Time Complexity:** The time complexity of the Huffman algorithm mainly depends on two operations: building the frequency table and constructing the Huffman tree. Building the frequency table involves scanning the input text once to count the frequency of each unique character, which takes $O(n)$ time, where n is the number of unique characters in the input text. Constructing the Huffman tree involves building a priority queue and repeatedly extracting the two nodes with the minimum frequencies until only one node remains. Since the priority queue needs to be sorted after each insertion, and there are n insertions, each of which takes $O(\log n)$ time, the overall time complexity of constructing the Huffman tree is $O(n \log n)$. Therefore, the overall time complexity of the Huffman algorithm is $O(n \log n)$.

Space Complexity: The space complexity of the Huffman algorithm mainly comes from building the frequency table and constructing the priority queue. Building the frequency table requires storing the frequency of each unique character in the input text, which results in $O(n)$ space complexity, where n is the number of unique characters. Constructing the priority queue involves creating nodes for each unique character, resulting in an additional $O(n)$ space complexity. Therefore, the overall space complexity of the Huffman algorithm is $O(n)$, where n is the number of unique characters in the input text.

2.1.4 *Algorithm*

. It constructs the tree by creating a binary heap where each node represents a character and its frequency. The algorithm repeatedly removes the two nodes of lowest frequency from the heap, combines them into a new node whose frequency is the sum of the two, and reinserts it back into the heap. This process continues until only one node remains, which represents the root of the Huffman tree, thus ensuring the shortest path for the most frequent characters and the longest path for the least frequent.

Algorithm 1 Huffman Coding

$n \leftarrow |c|$
$Q \leftarrow c$
for $i \leftarrow 1$ to $n - 1$ **do**
    $temp \leftarrow$ new node()
    $left[temp] \leftarrow$ Get_min(Q)
    $right[temp] \leftarrow$ Get_min(Q)
    $a \leftarrow left[temp]$
    $b \leftarrow right[temp]$
    $F[temp] \leftarrow f[a] + f[b]$
    Insert($Q, temp$)
end for
return Get_min(Q)

2.1.5 *Running Code*

. Here is the code:

```
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(text):
    freq = Counter(text)
    pq = [Node(char, freq) for char, freq in freq.items()]
```

```
    heapq.heapify(pq)
    while len(pq) > 1:
        left = heapq.heappop(pq)
        right = heapq.heappop(pq)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(pq, merged)
    return pq[0]

def build_huffman_codes(root):
    codes = {}
    def dfs(node, code):
        if node:
            if node.char is not None:
                codes[node.char] = code
            dfs(node.left, code + "0")
            dfs(node.right, code + "1")
    dfs(root, "")
    return codes

def encode(text, codes):
    encoded_text = ""
    for char in text:
        encoded_text += codes[char]
    return encoded_text

def huff(original_string):
    root = build_huffman_tree(original_string)
    huffman_codes = build_huffman_codes(root)
    encoded_string = encode(original_string, huffman_codes)
    return encoded_string
```

2.1.6 *Drawback*

. Huffman coding's main drawback lies in scenarios where symbol frequencies are highly skewed. In such cases, the resulting code may not achieve optimal compression, leading to less efficient use of space. Moreover, constructing the Huffman tree necessitates scanning the entire input, making it computationally intensive for large datasets. This limits its practicality in real-time or resource-constrained environments.

2.1.7 *Analysis of the Algorithm:*

The graph shows a positive correlation between compression time and input length. This means that it takes longer to compress larger datasets. This is because Huffman encoding involves building a tree that represents the probability of each symbol in the data. The larger the dataset, the more time it takes to build this tree. The graph also shows that the compression time is not constant. It appears to be increasing at an increasing rate. This is likely because the time it takes to build the Huffman tree grows faster than the size of the input data. For example, if the input data is twice as large, it may take four times as long to build the Huffman tree.

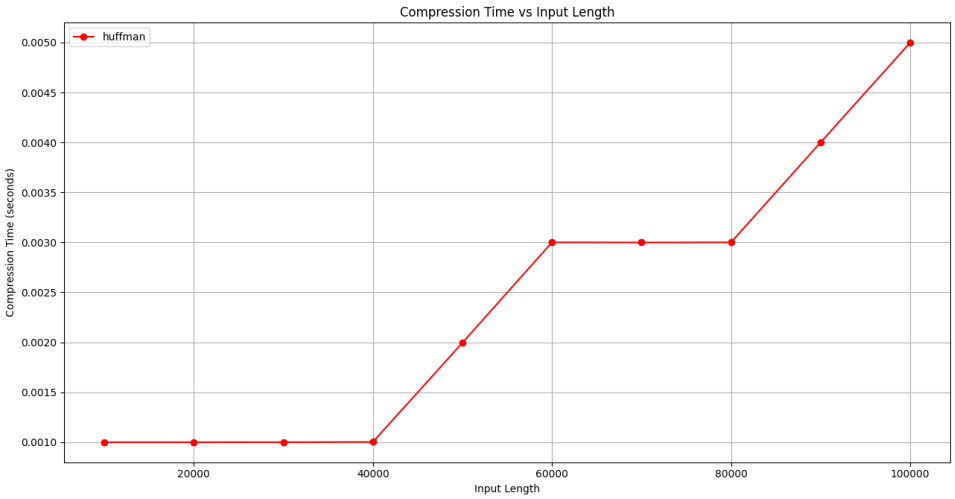

Fig. 1. Performance of the huffman encoding on Datasets Ranging from 10k to 100k

2.2 Run Length Encoding Compression (Greedy Technique)

2.2.1 *Introduction*

. Run-Length Encoding (RLE) is one of the fundamental techniques in data compression. RLE is one of the lossless techniques that is known for its simplicity and efficiency, as it can be used in various applications where data size reduction without loss of information is crucial. RLE operates on the principle of identifying consecutive sequences of identical data values, using these repetitive patterns to encode the data.

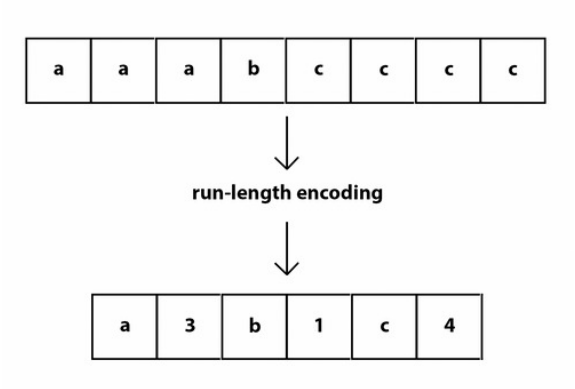

Fig. 2. Example of Run Length Encoding. We have bw(aaabcccc). Through this encoding, it is tranformed into bw(a3b1c4).

2.2.2 *Design Approach*

. RLE uses a simple method to identify the consecutive data values/characters and forms the pattern which are used to encode the data. Below we outline the critical components of its design:

- **Initialization and Management:** RLE begins with initializing an empty output sequence where the compressed data will be stored. As the algorithm processes the input, it adds new sequences that form, to form the encoding. We could also use a list or any other data structure to keep track of the encodings.
- **Compression Process:** RLE scans the input data sequentially, analyzing consecutive sequences of identical values, encoding them into representation of repeated value and frequency of the sequence. When it encounters a new character, that means the ongoing sequence is over, it updates the count and then moves on to the new sequence, repeating the process for the new sequence.
- **Decompression Process:** RLE's decompression process reverses the encoding to reconstruct the original data. It iterates over the compressed data, interpreting each pair of (value, count) to reconstruct the original sequence. For each pair, it appends the value to the output sequence count times, effectively replicating the original run of values.
- **Optimizations and Efficiency:** RLE's efficiency depends on the input data, particularly the presence of repeating patterns. The size of the output sequence may vary depending on the frequency and length of runs in the input data. For optimal performance, RLE implementations may include optimizations such as utilizing efficient data structures and algorithms for encoding and decoding processes, such as Huffman encoding and BWT for data compression.

From Figure 1, the steps for compression of the pixel are as follows:

2.2.1. Compression Process

a. Check of current value with the neighbors value, when current value same with the neighbors , then combine the values into one , and add counter to the values.

Step 1:

"(250,2)777725025025025025425411177777777733333325425455552502502502505552462462462462462467772462462462462502502502542543333355577777725425425425400000000001112542542502505555"

b. Go to the next value, if current value is not same with the neighbors value then save current value and repeat the first step back.

Step 2:

"(250,2)(7,4)2502502502502542541117777777773333325425455552502502502505552462462462462462467772462462462462502502502542543333355577777725425425425400000000001112542542502505555"

Step 3:

"(250,2)(7,4)(250,4)2542541117777777773333332542545555250250250255524624624624624624677724624624624625025025025425433333555777777254254254254000000000011125425425025055 5"

Step 26:

"(250,2)(7,4)(250,4)(254,2)(1,3)(7,5)(7,4)(3,6)(254,2)(5,4)(250,4)(5,4)(246,6)(7,3)(246,4)(250,3)(254,2)(3,5)(5,3)(7,6)(254,4)(0,10)(1,3)(254,2)(250,2)(5,3)"

Fig. 3. Example of RLE from source 1.
Note: The RLE algorithm used in the source focused more on integer/digits based encoding.

2.2.3 *Compression Algorithm*

. The Run Length Encoding compression algorithm starts by initializing an empty string. It then over the entire input. For every consecutive sequence encountered (such as *aaaa*), it adds one character of that sequence to the empty string and also adds the count to encode that part (*a*3). Here is the pseudocode for the RLE compression algorithm:

Algorithm 2 RLE compression

```
encoded_string ← empty
prev_char ← empty
count ← 0
while there is still data to read do
    current_char ← get next character from input string
    if current_char is equal to prev_char then
        count ← count + 1
    else
        if count > 0 then
            Add prev_char and count to encoded_string
        end if
        prev_char ← current_char
        count ← 1
    end if
end while
if count > 0 then
    Append prev_char and count to encoded_string
end if
```

2.2.4 *Running Code:*

Here is the code:

Listing 1. RLE Encoding Code

```
def run_length_encode(st):
    n = len(st)
    i = 0
    encoded_string = ""
    while i < n:
        count = 1
        if st[i].isdigit() == True:
            while i < n - 1 and st[i] == st[i + 1]:
                count += 1
                i += 1
            encoded_string += "(" + st[i] + ")" + str(count)
            i += 1
        else:
            while i < n - 1 and st[i] == st[i + 1]:
                count += 1
                i += 1
            encoded_string += "(" + st[i] + ")" + str(count)
            i += 1
    return encoded_string
```

2.2.5 *Theoretical Analysis:*

Time Complexity: The time complexity of the Run Length Encoding compression algorithm primarily depends on the length of the input data. The algorithm iterates through the entire input and checks each character in the input string once. For each character, it then performs comparisons and then either updates the encoded string or increases the count. The overall time complexity of the RLE compression algorithm: $O(n)$ time, where n is the length of the input text. This linear time complexity indicates that the time taken by the algorithm scales linearly with the size of the input data

Space Complexity: The space complexity of the RLE compression algorithm is $O(n)$, where n is the length of the input text. The space complexity mainly depends on the input data and the compressed output. In the worst case, where the input data contains no repeating sequences, the compressed output size may be twice the size of the input size, as each character and its count needs to be stored which means $2n$ hence $O(n)$. The best case, where the input data contains long consecutive sequences of repeated values, the compressed output size may be significantly smaller.

2.2.6 *Analysis of the Algorithm:*

The graph shows the compression time of the RLE algorithm for various datasets ranging in size from 10,000 to

100,000 elements. The x-axis represents the input length of the data (number of elements), and the y-axis represents the compression time in seconds.

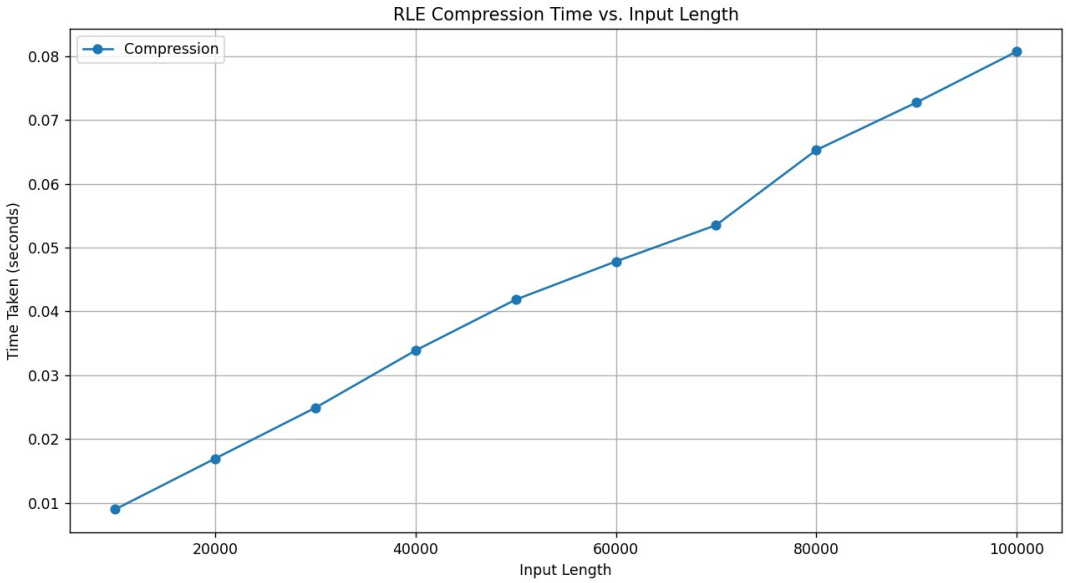

Fig. 4. Performance of the Run Length Encoding RLE Algorithm on Datasets Ranging from 10k to 100k

As the size of the input data increases, the compression time also increases. This is because the algorithm will take more time to process larger datasets as it iterates over the entire dataset to find the sequences to encode. One thing to note is that the graph doesn't show the actual compression ratio achieved by RLE, which is the ratio of the original data size to the compressed data size. While RLE is effective for compressing data with repeated sequences, it does not work best for data with random or unique values.

Overall, the graph confirms the expected behavior of RLE compression – the compression time increases as the size of the input data grows, a general characteristic of many compression algorithms.

2.2.7 *Drawbacks:*

Although Run-Length Encoding (RLE), is a simple and efficient compression algorithm, it has various limitations which can reduce its effectiveness. Probably the biggest significant drawback of this algorithm is experienced when it is given a diverse input data that lacks repetitive sequences. In such cases, RLE may fail to achieve significant compression ratios and might lead to an explosion in the size of the encoded string which might be greater than the original input. While RLE excels in compressing data with long runs of repeated values, its effectiveness whill reduce when the input lacks such patterns, resulting in limited reduction.. Furthermore, RLE also lacks contextual compression capabilities, as it solely compresses data on the basis of consecutive repetitions of characters. This lack of contextual awareness may further hinder it's workings.

2.3 Lempel–Ziv–Welch (LZW) Compression (Dictionary-Based Technique)

2.3.1 *Introduction*

. LZW is a dictionary-based compression and lossless technique, ensuring that the original data can be perfectly reconstructed from the compressed data. It always seeks to take the longest sequence of data it has seen before and encode it using the shortest possible code. The LZW algorithm is particularly effective for compressing data with repetitive patterns, making it ideal for text and image formats like GIF and TIFF. Its simplicity and efficiency stem from its ability to dynamically build a dictionary during the compression process, which optimizes compression without the need to transmit the dictionary. This reduces data transmission overhead and ensures data integrity, making it highly suitable for scenarios where accuracy is crucial.

2.3.2 *Design Approach*

. The Lempel-Ziv-Welch (LZW) algorithm employs a dynamic dictionary-based method for data compression and decompression, ensuring efficient handling and integrity of data. Below we outline the critical components of its design:

- **Dictionary Initialization and Management:** The process starts with a dictionary that includes all possible single-character strings from the input data's character set. As the algorithm processes the input, it adds new sequences that form, each assigned a unique index.
- **Compression Process:** The algorithm scans the input for sequences already in the dictionary, extending them by one character until it encounters a new sequence. It then outputs the index of the longest existing sequence and adds the new sequence to the dictionary.
- **Decompression Process:** Decompression utilizes the indices from the compressed data to accurately reconstruct the original data. It synchronizes with the compression dictionary, which allows for data integrity without the need to transmit the dictionary.
- **Optimizations and Efficiency:** The algorithm is highly effective for data with redundant or repeating patterns. The performance largely depends on the size and management of the dictionary. Enhanced implementations may incorporate technologies like content addressable memory (CAM) to accelerate dictionary look-ups, which is beneficial for speed in hardware implementations.

2.3.3 *Theoretical Analysis*

. **Time Complexity:** The time complexity for the LZW compression algorithm is expressed as $O(n + m + r)$, where n represents the length of the compressed data, m is the total size of the patterns, and r is the number of occurrences of these patterns. Generally, if the length of the compressed data n is much larger than the size of the patterns and their occurrences, the time complexity can be simplified to $O(n)$. This implies that the time taken by the algorithm is primarily dependent on the size of the data being processed.

Space Complexity: The space complexity is given as $O(m^2 + t + r)$, where m^2 refers to the square of the size of the patterns, t denotes the size of the dictionary used during compression, and r is related to the number of pattern occurrences. In scenarios where the patterns are complex or large, m^2 might dominate, simplifying the space complexity to $O(m^2)$. However, if the dictionary size t is significantly larger (common in scenarios where a full dictionary is utilized), the space complexity could effectively become $O(t)$. Thus, the dominant factor in the space complexity depends heavily on the specific characteristics of the data and the implementation details of the LZW algorithm.

2.3.4 *Compression Algorithm*

. This process starts with a dictionary that is pre-loaded with all possible symbols that can occur in the input data.

During compression, the algorithm scans through the input data and looks for the longest substring that matches an entry in the dictionary. When it finds a substring that is not in the dictionary, it outputs the code of the longest matching substring that is in the dictionary. Subsequently, it adds the new substring (including the last character that caused the mismatch) to the dictionary with the next available code. The next input sequence starts with the last unmatched character, continuing the process until all input data is processed. As the dictionary grows, the possibility of finding longer matching substrings increases, which improves the compression ratio over time.

Algorithm 3 LZW Compression

```
j ← input character
while there is still input character do
    ch ← transfer input string to ch
    if ch is in dictionary then
        Generate its codeword
    else
        Update ch and get next character to ch
        Search ch in dictionary
        if not present in dictionary then
            Add ch to dictionary
        end if
    end if
end while
```

2.3.5 *Running code*

. Here is the code:

```python
def lzw_encode(data):
    """LZW encoding for the input data."""
    dictionary = {chr(i): i for i in range(256)}
    p = ""
    compressed = []
    for c in data:
        pc = p + c
        if pc in dictionary:
            p = pc
        else:
            compressed.append(dictionary[p])
            dictionary[pc] = len(dictionary)
            p = c
    if p:
        compressed.append(dictionary[p])
    return compressed
```

2.3.6 *Analysis of the Algorithm:*

The graph shows a linear relationship between the input length and the compression time. This means that the compression time increases as the size of the dataset increases. For example, it takes about 0.005 seconds to compress a 20,000 element dataset, but it takes about 0.02 seconds to compress a 100,000 element dataset. This is because LZW encoding is a sequential algorithm. This means that it needs to process each element in the dataset one at a time. As the size of the dataset increases, the amount of time it takes to process all of the elements also increases.

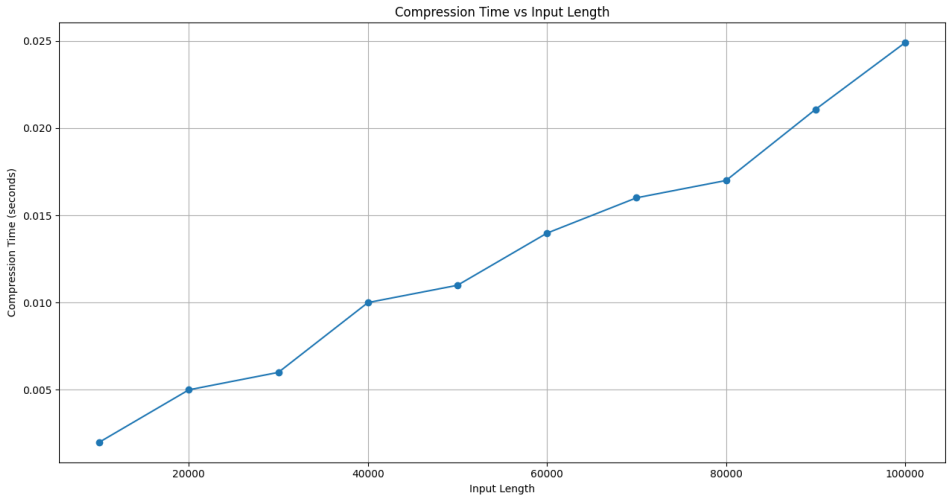

Fig. 5. Performance of the LZW encoding on Datasets Ranging from 10k to 100k

2.3.7 *Drawback*

. Its performance can decline with input data that has high diversity and fewer repetitive sequences, leading to rapid dictionary growth and potential overflow issues. Managing this overflow is crucial as it can either constrain the efficiency of the algorithm or necessitate a reset of the dictionary, which could interrupt the compression process. Furthermore, in environments where memory is limited, the expanding size of the dictionary can become problematic, requiring more sophisticated management strategies to balance compression effectiveness with practical resource constraints.

2.4 Burrows-Wheeler Transform (BWT) algorithm (Transformation Technique)

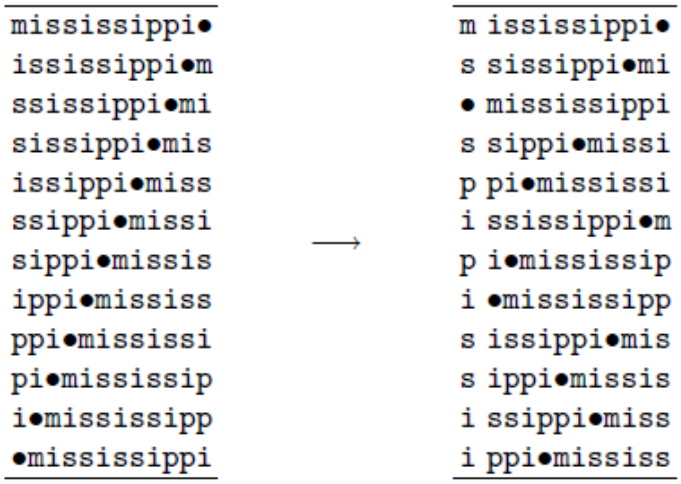

Fig. 6. Example of Burrows-Wheeler transform. We have bw(mississippi) = msspipissii. The matrix on the right is obtained by sorting the rows in right-to-left lexicographic order.

2.4.1 *Introduction:*

In the ever-expanding landscape of digital data, the quest for efficient data compression techniques remains crucial. The Burrows-Wheeler Transform (BWT) stands as a testament to this quest, offering a significant leap forward in lossless data compression. Unlike traditional methods, the BWT produces a permutation of the input string that is not only easier to compress but also allows for the original string to be retrieved. Its introduction marked a pivotal moment in the field of data compression, providing a transformative approach that significantly enhances compression efficiency. Even the simplest BWT-based algorithms have demonstrated impressive compression performances, surpassing commercially available packages like pkzip. Advanced BWT-based compressors, such as bzip2 and szip, have further cemented the BWT's position as one of the most powerful tools in lossless data compression.

2.4.2 *Design Approach of the Burrows-Wheeler Transform (BWT) Algorithm:*

- **Suffix Array Generation:** The first step in the Burrows-Wheeler Transform algorithm is to generate a suffix array for the input text. This involves appending an end-of-text marker '$' to the input text and then generating a suffix array using a sorting algorithm. This sorting algorithm sorts the suffixes of the text lexicographically, creating a sorted list of all suffixes.
- **Burrows-Wheeler Transform (BWT) Construction:** Once the suffix array is generated, the Burrows-Wheeler Transform (BWT) is constructed. This involves selecting the last character of each suffix in the sorted suffix array. The BWT is then constructed based on the sorted order of the suffixes, resulting in the transformed BWT.
- **Compression:** With the BWT constructed, the next step is to compress the transformed text. This compression step involves using the `bwt_compress` function to obtain the BWT of the input text. However, unlike some other

versions of the BWT algorithm, the algorithm version does not include additional encoding or compression techniques such as Move-to-Front (MTF) encoding or arithmetic coding (However the running code does.)

- **Compression Time Measurement:** To evaluate the efficiency of the compression process, the compression time is measured. This measurement is performed using the `measure_compression_time` function. First, the input text is read from a file to be compressed. Then, the time taken for compression is measured using the `time` module. This allows for the evaluation of the efficiency of the BWT compression process.

2.4.3 Compression Algorithm: The following Algorithm shows the working of the BWT

Algorithm 4 Burrows-Wheeler Transform (BWT)

```
text ← input text
text ← text + '$'
suffix_array ← SortSuffixes(text)
bwt ← ConcatenateLastCharacters(suffix_array, text)
return bwt
function SortSuffixes(text)
    suffixes ← [(i, text[i :]) for i ← range(len(text))]
    sorted_suffixes ← sorted(suffixes, key = lambdax : x[1])
    return [index for index, _ in sorted_suffixes]
end function
function ConcatenateLastCharacters(suffix_array, text)
    return ''.join(text[i − 1] for i ← suffix_array)
end function
```

2.4.4 Theoretical Analysis:

Time Complexity: The time complexity of the Burrows-Wheeler Transform (BWT) compression algorithm is dominated by the sorting step, which sorts the suffixes of the input text. Generating the suffix array takes $O(n \log n)$ time, where n is the length of the input text. Constructing the BWT from the sorted suffix array takes linear time, $O(n)$. Therefore, the overall time complexity of the BWT compression algorithm is $O(n \log n)$, where n is the length of the input text.

Space Complexity: The space complexity of the BWT compression algorithm is $O(n)$, where n is the length of the input text. This is because the algorithm requires additional space to store the suffix array, which has a size proportional to the length of the input text. Additionally, the algorithm constructs the BWT string, which also requires $O(n)$ space to store. Therefore, the total space complexity is $O(n)$.

2.4.5 Analysis of the Algorithm:

The graph shows the compression time of the BWT algorithm for various datasets ranging in size from 10,000 to 100,000 elements. The x-axis represents the input length of the data (number of elements), and the y-axis represents the compression time in seconds.

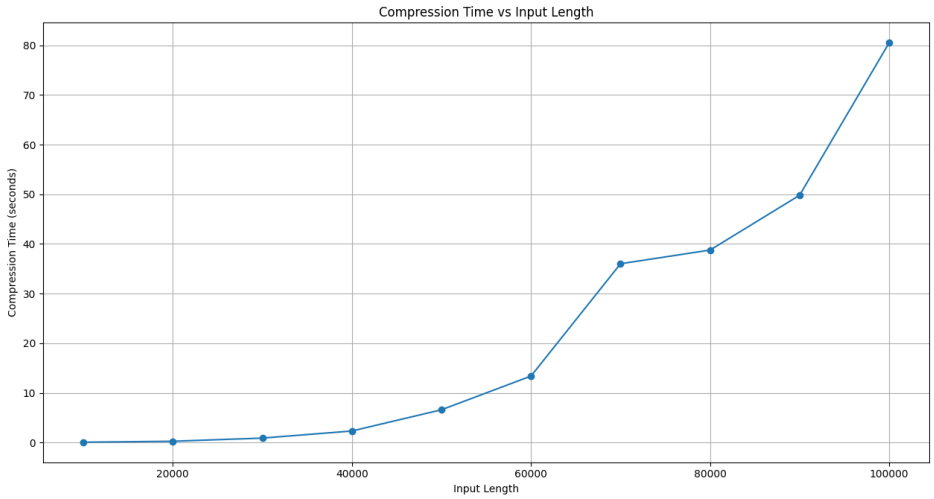

Fig. 7. Performance of the Burrows-Wheeler Transform (BWT) Algorithm on Datasets Ranging from 10k to 100k

As the size of the input data increases, the compression time also increases. This is because the BWT algorithm takes more time to process larger datasets. The increase in compression time appears to be linear according to the graph. This means that the compression time increases by a constant amount for every unit increase in the input size.

It is also important to note that the exact compression time will vary depending on the specific implementation of the BWT algorithm and the hardware it is running on. However, the graph provides a general idea of how the compression time scales with the input size.

2.4.6 *Drawbacks:*

Most BWT-based compressors process input files in blocks for memory control and limited error recovery. Larger block sizes lead to slower algorithms but better compression. The most time-consuming step is computing the transformed string $\mathrm{bw}(s)$. While some implementations sort prefixes of s, others sort suffixes. However, this choice minimally affects runtime and compression. Sorting all suffixes of a string s has been extensively studied due to its importance in string matching. Suffix arrays, an alternative to suffix trees, require less memory and can be computed in $O(n \log n)$ time.

2.4.7 *Running Code:*

Here is the code:

Listing 2. BWT Encoding Code

```
def bwt_compress(text):
    # Add end of text marker
    text += '$'
    # Generate suffix array
    suffix_array = sorted(range(len(text)), key=lambda i: text[i:])
    # Construct BWT by taking the last character of each suffix
    bwt = ''.join(text[i-1] for i in suffix_array)
    return bwt

# Move-to-Front (MTF) encoding
def mtf_encode(text):
    alphabet = list(sorted(set(text)))
    encoded_text = []
    for char in text:
        idx = alphabet.index(char)
        encoded_text.append(idx)
        del alphabet[idx]
        alphabet.insert(0, char)
    return encoded_text

# Run-Length Encoding (RLE) compression used mention in above section

# Compress text
def compress(text):
    bwt = bwt_compress(text)
    mtf = mtf_encode(bwt)
    rle = rle_compress(mtf)
    return rle
```

3 COMBINED EMPIRICAL PERFORMANCE AND RESULTS:

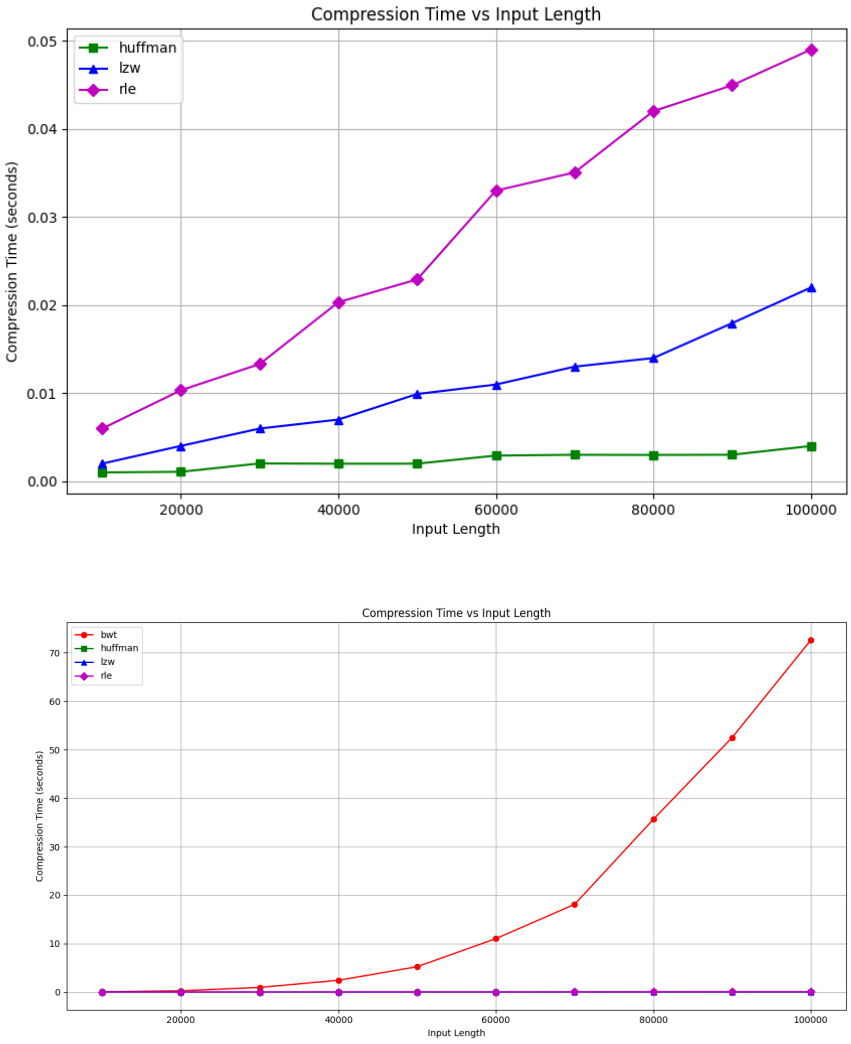

Fig. 8. Performance of 4 different Algorithms on same Datasets. The first graph compares the three Algorithms whose time can be compared while the second graph shows mainly the difference between BWT Algorithm with the other three Algorithms.

- **Huffman Encoding:** This approach proved to be the most efficient in terms of compression speed. It exhibits the lowest compression time across all tested input lengths. For instance, it takes about 0.002 seconds for a 10,000 character input, scaling up to approximately 0.008 seconds for a 100,000 character input. This efficiency makes it highly suitable for applications where quick data processing is critical, such as streaming audio and video where data needs to be compressed and decompressed in real-time.

- **RLE:** RLE demonstrates significant increases in compression time with larger datasets, indicative of its suitability for specific types of data. It begins at roughly 0.01 seconds for 20,000 characters and climbs to about 0.04 seconds at 100,000 characters.
- **LZW:** While not as fast as Huffman, LZW shows moderate efficiency. Its performance curve is steeper, indicating a faster increase in compression time with input size. It starts at about 0.005 seconds for 20,000 characters and progressively increases to approximately 0.02 seconds for 100,000 characters.
- **BWT:** It stands out as the slowest compression technique out the four algorithms. Its compression time is consistently higher across all data sizes from 10k to 100k characters. Moreover, the steep slope of BWT's curve in the graph indicates that its compression time increases much more rapidly than other techniques as input size grows. While BWT might excel in other areas like compression ratio, its time efficiency appears to be a significant drawback for the specified data range.

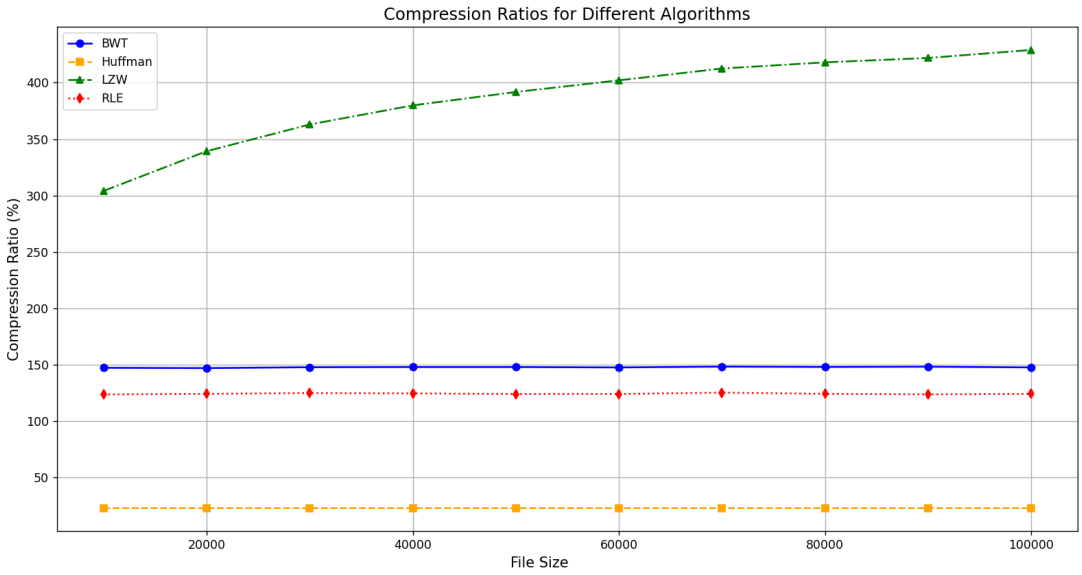

Fig. 9. Compression Ratios of the different algorithm on Datasets Ranging from 10k to 100k

The graph shows the compression ratios of the four algorithms. Although the compression ratio achieved by each algorithm can vary depending on the size of the original data, there are some findings evident from the graph. It is seen that Huffman does not Huffman encoding may not achieve significant compression which might be because the data contained a wide range of characters with relatively uniform frequencies. BWT perfomred better than RLE, as both were able to reduce the file size to a siginificant to a significant level. LZW performs the best which might be because of its dictionary-based approach, as it can identify and compress recurring patterns in the data.

4 CONCLUSION:

In conclusion, Huffman Encoding emerges as the fastest compression technique among the four algorithms evaluated, with the lowest compression time for all input lengths. LZW, while not as fast as Huffman, shows moderate efficiency with a gradual increase in compression time as input sizes grow, offering a good balance between speed and compression effectiveness suitable for general file compression tasks. RLE's performance, although increasing with larger datasets, remains relatively efficient and is particularly effective for data with repetitive sequences. In contrast, BWT exhibits a significant increase in compression time with larger input sizes, making it the least time-efficient among the tested algorithms. Although BWT may offer advantages in compression ratio, its time efficiency appears to be a significant drawback. In practical applications, the choice of compression algorithm involves a trade-off between compression ratio and compression speed. While Huffman Encoding is the fastest, it might not always provide the best compression ratio compared to algorithms like BWT. However, for applications where speed is crucial, such as real-time data transmission, Huffman Encoding would be the preferred choice, while BWT might be chosen when achieving the highest possible compression ratio is more important.

REFERENCES

[1] Manzini, Giovanni. "An Analysis of the Burrows—Wheeler Transform." Journal of the ACM, vol. 48, no. 3, May 2001, pp. 407–430, https://doi.org/10.1145/382780.382782.

[2] Aldwairi, Monther, et al. "MultiPLZW: A Novel Multiple Pattern Matching Search in LZW-Compressed Data." Computer Communications, vol. 145, Sept. 2019, pp. 126–136, https://doi.org/10.1016/j.comcom.2019.06.011.

[3] Kaur, Simrandeep. "Design and Implementation Af LZW Data Compression Algorithm." International Journal of Information Sciences and Techniques, vol. 2, no. 4, 31 July 2012, pp. 71–81, https://doi.org/10.5121/ijist.2012.2407.

[4] GeeksforGeeks. "LZW (Lempel–Ziv–Welch) Compression Technique - GeeksforGeeks." GeeksforGeeks, 26 Apr. 2017, www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/.

[5] GeeksforGeeks. "Huffman Coding | Greedy Algo-3." GeeksforGeeks, 3 Nov. 2012, www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

[6] DUDHAGARA, CHETAN R., and HASAMUKH B. PATEL. "Performance Analysis of Data Compression Using Lossless Run Length Encoding." Oriental Journal of Computer Science and Technology, vol. 10, no. 3, 10 Aug. 2017, pp. 703–707, https://doi.org/10.13005/ojcst/10.03.22.

[7] Hardi, Surya, et al. "Comparative Analysis Run-Length Encoding Algorithm and Fibonacci Code Algorithm on Image Compression." Journal of Physics, vol. 1235, no. 1, 1 June 2019, pp. 012107–012107, https://doi.org/10.1088/1742-6596/1235/1/012107.

[8] Huang, Junzhou. CSE5311 Design and Analysis of Algorithms 1 Dept. CSE, UT Arlington CSE5311 Design and Analysis of Algorithms 1 CSE 5311 Lecture 17 Greedy Algorithms: Huffman Coding Design and Analysis of Algorithms.