

David Zeff 207875147

Maha Elriati 208313197

Final Project Data Compression

Overview

Our project aims to develop a data compression algorithm that combines LZW (Lempel-Ziv-Welch) and Huffman coding. The goal is to create a more efficient compression method by using the frequency of sequences in the input text, this way the most frequent would be closest to the root making their bit sequence shorter, as opposed to common LZW where each sequence is given an index without considering frequency.

Initial Idea

Original Concept:

1) LZW-Based Dictionary:

We began by designing an algorithm that builds a dictionary like LZW. As the text is scanned, sequences are put into a table, with each sequence also being assigned a frequency value showing how often it appears.

2) Frequency-Based Huffman Tree:

After building the table, it is sorted by frequency. The idea was to then build a Huffman tree where the most frequent sequences would be closer to the root, than they would need fewer bits to encode.

3) Decoding:

After writing the Huffman tree and the compressed data to a file we could start decoding using standard Huffman practice of reading the byte sequence and encoding every sequence by traversing every bit/branch of the tree until reaching a leaf which would represent our decoded sequence (the sequence of bits we needed to traverse to the leaf would represent the coded sequence).

Challenges:

Huffman Tree Size:

The Huffman tree became too large, leading to inefficiencies. Searching for each sequence in the tree took too long (to the point where it was un-runable), with a Huffman tree that gets built from a very large group of sequences such as that of a sequences generated from a LZW algorithm, and as we know a Huffman tree needs to be built with a prefix tree condition so even in the best case searching the tree would take $\log(n)$ and this is assuming the height of the tree is $\log(n)$ which is incredibly unlikely, the actual height of a Huffman tree is probably closer to $O(n)$ making scanning the text plus searching the tree be n^2 time complexity making the code un runable, also the size of the tree was very large, consider that each node had a parent and children pointers and at least a byte for the code and on average a few bytes to represent the sequence.

Evolution:

Simplified Encoding Table:

Instead of using a Huffman tree, we created an index-based table. Sequences were assigned indices based on their frequency, with more frequent sequences receiving smaller indices, mimicking the principle of shorter bit sequences in Huffman coding, this is why it hasn't completely strayed from the original idea, but this time the code is runnable because we use a hash table to search for each sequence which take constant time.

Final Approach

Final Compression Algorithm:

1. LZW with Frequency Table:(see figure 1 below)

We run the LZW algorithm and build a frequency table for the sequences.

The table is then sorted, and a new table is created:

The first 256 entries correspond to ASCII characters (0-255).

After 255, the sequences are inserted based on their frequency, with the most frequent sequence receiving index 256, then 257, and so on.

Table Size Limitation: :(see figure 2 below)

To ensure effective compression, we limit the size of the table. This avoids the issue of the table size being too large where compression is not smaller than the original.

Example for step 1:

| sequence | frequency |
|----------|-----------|
| "ba" | 2 |
| "ab" | 3 |

| | |
|-------|---|
| "ban" | 3 |
| "na" | 1 |

Sort table:

| sequence | frequency |
|----------|-----------|
| "ab" | 3 |
| "ban" | 3 |
| "ba" | 2 |
| "na" | 1 |

Re-index the table:

| sequence | index |
|----------|-------|
| "ab" | 0 |
| "ban" | 1 |
| "ba" | 2 |
| "na" | 3 |

The actual table will have 256 acii characters at indexes 0 -255

2. Encoding Process: (see figure 3 below)

We encode the text using the table generated by LZW. For each sequence, we:

Check the number of bytes needed to represent the index.

Encode the number of bytes needed, followed by the sequence's index.

Single Character Flag: If the prefix is greater than 8, it indicates a single character, which we encode directly without using a prefix.

Optimization(NOT IMPLEMENTED DUE TO EDGE CASES(but still worth mentioning I think)):

We subtract 2 from the prefix value during encoding and add 2 during decoding. We can do this because we won't have any prefixes of 1 or 0 because any single byte sequences will be characters which will be encoded directly, and 0 prefix will not exist because we won't have a sequence that requires 0 bytes to represent this is why we can subtract 2 from every prefix. This allows for more efficient use of the bit space and provides room for future algorithm enhancements.

Example for step 2:

Given sequence: "ba"

1. Look for "ba" in table

- a. If "ba" is in table encode it:
 - "ba" index = 500
 - 500 requires 2 bytes so prefix is: 0000 0010(2), insert the bin as first byte
 - 500 = 0000 0001 1111 0100, encode the next 2 bytes as that bin

3. Handling Sequences Not in the Table: (see figure 4 below)

If a sequence is not found in the table, we encode each letter of the sequence individually.

Potential Improvement:

A more advanced method would be recursively finding subsequences within the sequence, but this was not implemented because it was too complex to code up and to many bugs and issues.

Example for step 3:

- b. if "ba" not in the table then encode each character individually:

-b = 0110 0010 (98) encode b

-a = 0110 0001 (97) encode a

4. Decoding Process (see figure 5 below)

The first byte in encodedByteArray is checked to figure out if it represents a single character or a sequence.

If the byte size is greater than 8, it shows that it's a single character, which is directly decoded and appended to the result.

If the byte size is 8 or less, it represents the number of bytes required to decode a sequence.

The value is reconstructed by shifting and combining the subsequent bytes into a single integer. This value is used as an index to get the corresponding sequence from sequenceTable, which is then appended to the decoded result.

The process continues until the entire encodedByteArray is decoded, and the result is returned as a string.

Example for step 4:

-First byte is 0000 0010(2) < 8, so it is a prefix so read two more bytes

-2nd and 3rd bytes: 0000 0001 1111 0100(500) search table at index 500

-At index 500: "ba"

-Append "ba" to the decoded file

- read next byte

-byte read is 0110 0001

-(0110 0001) (97) > 8 so it's not a prefix so encode it directly as character

-0110 0001 = 97 so append 'a' to the decoded file

Compression Example

File:

A text file with 25.5 million characters (25.8 MB). (see figure 6 below)

Results:

1. Compressed Output:

The process takes 13 seconds.

The resulting compressed file and sequence table total 21.5 MB, achieving about 18% compression.

Table Size Limitation Impact: The limited table size is the reason for the moderate compression rate. Without this limit, we could achieve an 80% compression rate, but at the cost of a significantly larger table size. (see figure 7 below)

2. Challenges in Table Size:

The sequence table generated without size limits is almost twice the size of the original text, making the compression ineffective. A method to recreate the table during decompression, like LZW, would address this issue because then we wouldn't have to send the table along and the only thing being sent would be the actual compressed data.

If we compress with no limit set on table size, we will achieve compression of 80% but the table would be too large (see figure 8 below)

Comparison table with Standard LZW

| Algorithm/File | Romeo.txt (159 KB) | RomeoFullPlay.txt (25.8 MB) |
|----------------|--|--|
| My LZW | Compressed: 93 KB, Sequence Table: 293 KB, Total: 386 KB | Compressed: 20.8 MB, Sequence Table: 800 KB Total: 21.6 MB |
| Standard LZW | 142 KB | 6.65 MB |

GUI

(see figure 9 below)

We can select a text file to compress, with the compressed file path, decompressed file path, and sequence table path automatically generated based on the original text file's location.

Execution:

We can also Compress and Decompress separately, meaning if we have a compressed file along with its sequence table, then we can select them and enter “decompress” without compressing them.

Standard LZW Class(Used for comparison)

We used a Standard LZW algorithm to compare to our “Advanced” version.

It's based off the pseudo code given by Professor Bill Bird on YouTube but implemented in java (also the LZW related code in the advanced is based off him).

Compression

- 1.Read the text from the file.
- 2.Initialize the symbol table with ascii characters and then use the lzw algorithm to build the table, every sequence will be encoded into a integer list based on its code and then we will convert this list to a byte array which will be our encoding.
- 3.write the encoded byte array to a file(this is our compressed txt).

Decompression

- 1.Read the byte array and convert it to an integer list so that it will be easier to work with.
- 2.Rebuild the dictionary while decoding the codes using the dictionary being build jit., using LZW algorithm.

3. write the decoded txt to a file

Figures

Figure 1(generating sequence table: generate frequency table->sort the table->re define the indices)

```
public Map<String, Integer> generateSequenceTable(String rawTextFromFile) throws IOException {
    // create a map that holds all sequences along with their respective frequency
    Map<String, Integer> freqMap = calculateFrequency(rawTextFromFile);
    // this will store the final sequences that we will use to decode in
    // decompression
    Map<String, Integer> sequenceTable = new LinkedHashMap<>();

    // initialize the most common chars to have the lowest indices(all single
    // character ascii values)
    for (int i = 0; i < 256; i++) {
        sequenceTable.put("" + (char) i, i);
    }

    int index = 256;
    // for every sequence put it into the table with its respective rank is its
    // index, this way the most common sequences will have the lowest index and
    // shortest byte sequence
    for (String sequence : freqMap.keySet()) {
        if (!sequenceTable.containsKey(sequence)) {
            sequenceTable.put(sequence, freqMap.get(sequence) + index);
            index++;
            // limit the size of our table
            if (maxSizeOfTable <= index) {
                break;
            }
        }
    }

    return sequenceTable;
}
```

Figure 2(the table size limit 20 * 2024 entries)

```
compressionObj = new AdvancedLZW2(unCompressedFile.getText(), unCompressedFile.getText(),
    decompressedFile.getText(), sequenceTableFile.getText(), 20 * 2024);
```

Figure 3(encoding process)

```

private List<Byte> encodeSequence(String sequence, Map<String, Integer> sequenceTable) {
    List<Byte> encoded = new ArrayList<>();

    // check if sequence is in sequence table
    if (sequenceTable.containsKey(sequence)) {
        // here we get the value and the amount of bytes needed to represent the value
        int value = sequenceTable.get(sequence);
        String binStr = Integer.toBinaryString(value);
        int bitLength = binStr.length();
        int byteSize = (bitLength + 7) / 8;

        // if byteSize is 1 and the value is <=255 then it is a single character that we
        // will encode
        // directly without a prefix because when we decode we will know that its a
        // single char because its a byte size greater then lets say 8
        // and we know that its practically not possible that we will have a sequence in
        // the table that will need 8 bytes in order to represent it
        if (byteSize == 1 && value <= 255) {
            encoded.add((byte) value);
        } else {
            // else it is a sequence that will have a prefix to represent the amount of
            // bytes needed for the code(will definetly be less then 8 practically speaking
            // this is how we will know its a sequence not a char)
            encoded.add((byte) (byteSize));

            // Add each byte of shifting to the right byte in the value and inserting that
            // byte and then shifitng until we inserted the whole value as a byte sequence
            for (int i = byteSize; i > 0; i--) {
                int shiftAmount = (i - 1);
                shiftAmount *= 8;
                byte currentByte = (byte) (value >>> shiftAmount);
                encoded.add(currentByte);
            }
        }
    } else {
        // if sequence not in table encode each char independently
        // directly without a prefix because when we decode we will know that its a
        // single char because its a byte size greater then lets say 8
        // and we know that its practically not possible that we will have a sequence in
        // the table that will need 8 bytes in order to represent it
        char[] charsInSequence = sequence.toCharArray();

        for (char c : charsInSequence) {
            int charValue = sequenceTable.get(String.valueOf(c));
            encoded.add((byte) charValue);
        }
    }

    return encoded;
}

```

Figure 4(handling a sequence not in the table)


```

else {
    // if sequence not in table encode each char independently
    // directly without a prefix because when we decode we will know that its a
    // single char because its a byte size greater then lets say 8
    // and we know that its practically not possible that we will have a sequence in
    // the table that will need 8 bytes in order to represent it
    char[] charsInSequence = sequence.toCharArray();

    for (char c : charsInSequence) {
        int charValue = sequenceTable.get(String.valueOf(c));
        encoded.add((byte) charValue);
    }
}

```

Figure 5(decoding process)

```

private String decodeWithSequenceTable(byte[] encodedByteArray, Map<Integer, String> sequenceTable) {

    StringBuilder decoded = new StringBuilder();

    int i = 0;

    // loop through the encoding and decode the bytes
    while (i < encodedByteArray.length) {

        int byteSize = encodedByteArray[i];

        // this is a flag that tells us it must be a single char because practically we
        // will not have a sequence that has a code that requires more than 8 bytes, if
        // there is a sequence that has a value that is
        // more than 8 bytes it means its index 1 with 19 0s, this is just not
        // practical(not to mention that we limit our table size anyway). so its ok to
        // use 8 as a flag.
        // the only issue is we will miss out on chars such as the first 8 ascii chars
        // but for now its ok since we are just parsing regular text and
        // not anything that we will expect to be parsing those first 8 ascii chars
        if (byteSize > 8) {

            char oneChar = (char) byteSize;
            decoded.append(oneChar);
            i++;
        } else {

            int value = 0;
            //look at the readme to understand this, we explain it there
            byteSize++;
            for (int j = 0; j < byteSize; j++) {
                // what we do here is the inverse of what we did when we encoded,we inserted the
                // value
                // using byte sequences, now we want to insert the byte sequences into 1
                // value,before we divided the value into bytes
                i++;
                // make space for next byte (we are turning the byte sequence into its original
                // value it was divided from during encoding)
                value = value << 8;
                // and ff is to make sure its unsigned otherwise it will cause issues
                int unsignedByte = encodedByteArray[i] & 0xFF;

                value = value | unsignedByte;
            }

            String sequence = sequenceTable.get(value);

            decoded.append(sequence);

            i++;
        }
    }

    // Convert the StringBuilder to a String and return the decoded result
    return decoded.toString();
}

```

Figure 6(file to be encoded)

Romeo and Juliet Entire Play.txt

FileEditView

Romeo and Juliet

Shakespeare homepage <<http://shakespeare.mit.edu/Shakespeare>> | Romeo and Juliet <http://shakespeare.mit.edu/romeo_juliet/> | Entire play

ACT I

PROLOGUE

Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth the fatal loins of these two foes
A pair of star-cross'd lovers take their life;
Whose misadventured piteous overthrows
Do with their death bury their parents' strife.
The fearful passage of their death-mark'd love,
And the continuance of their parents' rage,
Which, but their children's end, nought could remove,
Is now the two hours' traffic of our stage;
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.

Ln 1, Col 125,454,401 characters

Figure 7(compressed with table size limit set to 2 * 2024)



| | | | | |
|---|---|------------------|---------------|-----------|
|  RomeoCompressed |  | 11/08/2024 22:33 | Text Document | 20,839 KB |
|  SequenceTable |  | 11/08/2024 22:33 | Text Document | 813 KB |

Figure 8(compressed with no table size limit set)

| | | | | |
|---|---|------------------|---------------|-----------|
|  RomeoCompressed |  | 11/08/2024 22:37 | Text Document | 4,373 KB |
|  SequenceTable |  | 11/08/2024 22:37 | Text Document | 44,632 KB |

Figure 9(GUI)

