Ruth Tilahun
CMSC 337
Lab 10

Data Compression: LZW Algorithm

Introduction

Data compression refers to the process of reducing the size of data by encoding it using fewer bits than the original representation. This is useful for reducing storage requirements and transmission times. There are two main categories of compression algorithms: lossless and lossy. Lossless compression allows the original data to be perfectly reconstructed from the compressed version while lossy compression accepts some data loss in exchange for higher compression ratios. In this report, we describe an implementation of the LZW(Lempel-Ziv-Welch) algorithm, which is a dictionary-based approach to lossless data compression. It takes advantage of repetitive patterns of data so that once these patterns are identified, they can be encoded into a compact code symbol. This symbol is then used to represent the recurring pattern throughout the file.

LZW Compression Algorithm

The LZW algorithm is a greedy algorithm that operates by trying to recognize increasingly longer and longer phrases that are repetitive in the input data and encode them using a dictionary. Each new phrase that is added to the dictionary is defined as having a prefix that is equal to a previously encoded phrase, with one additional character from the ASCII character set appended to it.

In Figure 1, the algorithm starts by initializing a dictionary with all single character strings mapped to their ASCII values (0-255). So, the next new phrase – formed from a sequence of strings – will be stored in the dictionary starting from the 256th index with the index representing the next available code.

```java
Map<String, Integer> dictionary = new HashMap<>();

for (int i = 0; i < 256; i++) {
     dictionary.put("" + (char) i, i);
}

// List to store the output indices
List<Integer> output = new ArrayList<>();
int dp = 256; // Next index to add new strings
```

Figure 1: Java code snippet for implementation of LZW algorithm

```
String s = "" + text.charAt(0);
for (int i = 1; i < text.length(); i++) {
    char c = text.charAt(i);
    String sc = s + c;

    if (dictionary.containsKey(sc)) {
       s = sc; // If sc is in the dictionary, continue with sc
    } else {
         // Output the code for s
         output.add(dictionary.get(s));
         // Add sc to the dictionary
         dictionary.put(sc, dp++);
         // Start new string s with character c
         s = "" + c;
     }
  }
```

Figure 2: Java code snippet for implementation of LZW algorithm

In Figure 2, the compression algorithm builds the dictionary in a greedy fashion by reading through the input data character-by-character. At each step, it checks if appending the current character, *c,* to the active phrase, *s,* results in a new phrase that is already present in the dictionary. If so, it continues extending the active phrase. If not, it outputs the code for the current active phrase, adds the new extended phrase to the dictionary, and starts over with the current character as the new active phrase. Thus, LZW is able to identify and encode the longest possible repetitive sequences by iteratively expanding the active phrase until it no longer matches any dictionary entries.

For example, let's encode the text "ABABC". First, we initialize the active phrase *s* to the first character 'A'. The algorithm then reads the next character 'B' and since *sc* = "AB" is not yet in the dictionary, it outputs the code for 'A' (65), adds "AB" to the dictionary as code 256, and resets *s* to "B". Reading 'A' next, again, since "BA" is not in the dictionary, it outputs 66 for 'B', adds "BA" as code 267 to the dictionary, and sets *s* to "A". It then reads 'B', making *sc* = "AB", which we just added as code 256 in the dictionary. Now, since the algorithm has encountered a phrase that it has seen before, it will continue to add the next character on it until it forms a phrase that it has not seen before/is not in the dictionary. Thus, continuing, it reads 'C' which makes *sc* = "ABC". But "ABC" is a new phrase/not in the dictionary yet, so it outputs the code for "AB" (256), adds "ABC" as code 258 and sets s to "C". The dictionary now contains codes up to 258 for repeated phrases found in the input text.

The runtime complexity of the LZW compression algorithm is $\Theta(n)$. Initializing the dictionary with takes $\Theta(1)$ time since there are only 256 single-character strings. The main loop performs constant time operations when doing dictionary lookups. In the worst case where a character sequence isn't in the dictionary, the algorithm adds a new entry in Θ(1) time on average since it is HashMap operation. The loop iterates over each character in the input text once, so the total time complexity of the loop is Θ(n), where n is the length of the input text. Therefore, LZW compression has an overall linear time complexity of Θ(n).

LZW Decompression Algorithm

The LZW compression and decompression algorithms in a complementary way. During compression, the algorithm builds the dictionary dynamically by adding new phrases as it encounters them in the input text. During decompression, the algorithm reconstructs the same dictionary by following the same rules used during compression, allowing it to correctly map the codes back to their corresponding phrases.

The *decompress* method in Figure 3 converts a list of *indices* – compressed codes outputted from the LZW compression algorithm – back into the original string.

```
 public String decompress(List<Integer> indices) {
        int dictSize = 256;
        Map<Integer,String> codes = new HashMap<Integer,String>();
        for (int i = 0; i < 256; i++)
            codes.put(i, "" + (char)i);

        StringBuffer result = new StringBuffer();
        int c = indices.remove(0);
        String previous = codes.get(c);
        result.append(previous);
        for (int current : indices) {
            String s;
            if (codes.containsKey(current))
                s = codes.get(current);
            else if (current == dictSize)
                s = previous + previous.charAt(0);
            else
                throw new IllegalArgumentException("Bad compressed 
current: " + current);
            result.append(s);
            codes.put(dictSize++, previous + s.charAt(0));
            previous = s;
        }
        return result.toString();
    }
```

Figure 3: Java implementation of LZW decompression algorithm

Initially, the algorithm populates a dictionary (HashMap) *codes* with mappings for all single character strings corresponding to the ASCII range from 0 to 255. It then reads the first code from the input list, retrieves its corresponding string from the dictionary and stores the resulting decompressed output in *result.* The algorithm can assume that the first code would represent a single character since the compression procedure starts with existing characters and its dictionary is also initialized with all single characters. Handling the first code separately gives access to the *previous* variable which holds the last decoded string and is used to reconstruct the subsequent strings in the loop.

Inside the loop, for each code, the algorithm checks if the code exists in the dictionary. If it does, it retrieves the corresponding string and appends it to *result.* If the code is not in the dictionary, the algorithm assumes that it is the next available dictionary index/code *dictSize* (which starts off at 256 and is incremented after each code that isn't in the dictionary is encountered). To construct the string for this code, it appends the first character of the *previous* string to itself(*s = previous + previous).* Then, the algorithm updates the dictionary by appending the first character of the current string to the previous string(*s.charAt(0) + previous)* and adding it to the proper index, *dictSize.* After processing each code, *result* will contain the fully decompressed string.

The LZW decompression, like the compression, has a runtime complexity of Θ(n) where n is the length of the input list of compressed codes. The algorithm iterates over the input list once and performs constant time operations for dictionary lookups and insertions on average.

Discussion

	Size of Text (in bytes)	Output of Compress (in bytes)	# Entries in Dictionary (compress)	Output of Decompress (in bytes)
Sample Text 1	53	156	294	53
Macbeth	149	464	371	149
Moby Dick	1220079	938572	234898	N/A[1]

Figure 4: Compression performance results on sample tests

As shown in Figure 4, the LZW compression doesn't always result in a compression. For small inputs like the two short sample texts, the compressed data is larger than the size of original text. This is because the algorithm has

[1] The provided decompression code did not work for Moby Dick, it kept on going forever.

some inherent overhead/extra information it needs to store about the dictionary so that the decompressor can reconstruct codes into phrases. For small inputs which tend to have little redundancy, the overhead ends up being larger than any compression gained. On the other hand, for larger inputs with significant repetition of phrases, like Moby Dick, the ability to represent those repeated strings as shorter codes outweighs the dictionary overhead cost, resulting in an overall compressed file that is smaller than the original(1.2MB to 0.9MB). One possible way to handle this issue is to modify the algorithm to output the original character codes for inputs below some size threshold without building the dictionary to ensure that there is enough redundancy.

Additionally, this implementation only handles ASCII characters in the 0-255 range. If there were characters outside of this range in the input, they would likely be ignored or misinterpreted since the initial dictionary is populated only with entries for the 256 characters. This would lead to inconsistencies during compression and decompression.

Finally, the sizes of the decompressed strings are equal to their original sizes, proving that LZW is a lossless compression algorithm.

Conclusion

The LZW compression algorithm works by greedily extending a phrase until adding the next character would create a new phrase not yet in the dictionary. LZW is able to encode a string by outputting codes for prefixes already in the dictionary along with adding new extended phrases to the dictionary. The effectiveness of LZW compression depends heavily on having sufficient redundancy and repeated patterns in the input data to leverage the dictionary compression approach.

The LZW compression and decompression algorithms work together by sharing the same dictionary initialization and using complementary rules for building and utilizing the dictionary. This relationship ensures lossless data compression and decompression.

Appendix: Java Code

```
public class lzw {
    public static List<Integer> lzwCompress(String text) {
        // Dictionary to store the data, initially with all single
characters (ASCII)
        Map<String, Integer> dictionary = new HashMap<>();
        for (int i = 0; i < 256; i++) {
            dictionary.put("" + (char) i, i);
        }

        // List to store the output indices
        List<Integer> output = new ArrayList<>();
        int dp = 256; // Next index to add new strings

        String s = "" + text.charAt(0); // Start with the first
character
        for (int i = 1; i < text.length(); i++) {
            char c = text.charAt(i);
            String sc = s + c;

            if (dictionary.containsKey(sc)) {
                s = sc; //If sc is in the dictionary, continue with sc
            } else {
                // Output the code for s
                output.add(dictionary.get(s));
                // Add sc to the dictionary
                dictionary.put(sc, dp++);
                // Start new string s with character c
                s = "" + c;
            }
        }
        // Output the code for the last value of s
        output.add(dictionary.get(s));

        int textSixe = text.length(); // each character is 1 byte
        int compressedSize = output.size() * 4; // each integer is 4
bytes
        int entries = dictionary.size();
        System.out.println("Text size: " + textSixe + " bytes");
        System.out.println("Compressed size: " + compressedSize + "
bytes");
        System.out.println("Dictionary entries: " + entries);
        return output;
    }
```