Programming Project: LZW Compression

Introduction

The Lempel–Ziv–Welch (LZW) algorithm is a lossless data compression algorithm. LZW is an adaptive compression algorithm that does not assume prior knowledge of the input data distribution. This algorithm works well when the input data is sufficiently large and there is redundancy in the data. Two examples of commonly used file formats that use LZW compression are the GIF image format served from websites and the TIFF image format. LZW compression is also suitable for compressing text files, and is the algorithm in the *compress* Unix file compression utility.

This algorithm has two steps:
1. Encoding/Compressing
2. Decoding/Decompressing

The interesting thing is that the encoding table, or dictionary, computed during the encoding process does not need to be explicitly transmitted. It can be regenerated from the coded/compressed data.

Encoding/Compressing

The LZW algorithm reads an input sequence of symbols, groups the symbols into strings, and represents the strings with integer codes. Since the codes require less space than the strings, compression is achieved. LZW is a greedy algorithm in that it finds the longest string that it has a code for, and then outputs that code. LZW starts out with a known dictionary of single characters that constitute the input character set. For example, this could be the extended ASCII set of 256 characters (for the case of 8 bits) that represent alphabets, numbers, punctuation symbols, etc., and it uses these as the "standard" character set (see www.asciitable.com). It then reads symbols (i.e., characters) 8 bits at a time (e.g., 't', 'h', 'e',etc.) and appends them one by one to the current string. Each time it appends a symbol to the string, it checks whether the resulting string has a code in the dictionary. If it does, it reads in the next symbol and appends it to the current string. If the resulting string does not exist in the dictionary, it means it has found a new string: it outputs the code corresponding to the string without the newest symbol, adds the string concatenated with the newest symbol (i.e., the new string) to the dictionary with its code (which is the previous largest code value incremented by one), and resets the current string to the newest symbol. Thus the next time the LZW algorithm encounters a repeated string in the input sequence, it will be encoded with a single number. The algorithm continues to process symbols from the input sequence, building new strings until the sequence is exhausted, and it then outputs the code for the remaining string. Usually a bit length for the codes is specified, based on the maximum number of entries for the dictionary, so that the process does not use a very large amount of memory. So when the codes representing the strings are 12 bits long, the table size is 2^{12} = 4096. It is necessary for the bit length of the codes to be longer than that of the characters (12 bits vs. 8 bits), but since repeated strings in the input sequence will be replaced by a single code, compression is achieved. LZW adaptively builds the dictionary based on the input sequence.

Here is the pseudocode for the LZW encoding algorithm:

```
MAX_TABLE_SIZE=2^(bit_length)   //bit_length is number of encoding bits
initialize TABLE[0 to 255] = code for individual characters
STRING = null
while there are still input symbols:
    SYMBOL = get input symbol
    if STRING + SYMBOL is in TABLE:
        STRING = STRING + SYMBOL
    else:
        output the code for STRING
        If TABLE.size < MAX_TABLE_SIZE:   // if table is not full
            add STRING + SYMBOL to TABLE // STRING + SYMBOL now has a code
        STRING = SYMBOL
output the code for STRING
```

The bit length of the output is a user defined parameter, usually in the range 9 to 12. The table size, defined in the pseudocode as MAX_TABLE_SIZE, depends on the bit length, and is $2^{(\text{bit length})}$.

Note that every time the algorithm adds a new string and code to the dictionary, it also outputs a code. Also, as the dictionary grows, the lengths of the strings it holds increases. Finally, the encoding table, or dictionary, computed during the encoding process does not need to be explicitly transmitted.

Decoding/Decompressing

Decompression works in the reverse fashion to compression, converting integer codes into the strings they represent. The decompression process for LZW is also straightforward to code, although a little more involved to understand. In addition, it has an advantage over static compression methods because no dictionary or other overhead information is necessary to be transmitted for the decoding algorithm. A dictionary identical to the one created during compression is reconstructed during the decompression process. Both encoding and decoding programs must start with the same initial dictionary, in this case, all 256 extended ASCII characters.

Here is how it works. Mirroring the process carried out by the encoder, every time the decoder extracts a string from the dictionary using a code, it adds a string to the dictionary, consisting of the previous string and the first character of the new string, with an updated code index. So the decoder is adding strings to the dictionary one step behind the encoder.

The LZW decoder first reads an input code (integer) from the encoded sequence, looks up the code in the dictionary by using it as an index, and outputs the string associated with the code. Thereafter, the decoder reads in a new code, finds the new string indexed by this code, and outputs it. The first character of this new string is concatenated to the previously decoded string. This new concatenation is added to the dictionary with an incremented code (simulating how the strings were added during compression). The decoded new string then becomes the previous string, and the process repeats.

When the decoder receives a code that is not already in its dictionary, it can be shown that the new string corresponding to this code consists of the previously decoded string with the first character of the previously decoded string appended. Each time it reads in a code that does not exist in the dictionary, it must add the code and the corresponding string to the dictionary.

Here is the pseudocode for the LZW decoding algorithm:

```
MAX_TABLE_SIZE=2^(bit_length)
initialize TABLE[0 to 255] = code for individual characters
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING
while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined:      // needed because sometimes the
        NEW_STRING = STRING + STRING[0] // decoder may not yet have code!
    else:
        NEW_STRING = TABLE[CODE]
    output NEW_STRING
    if TABLE.size < MAX_TABLE_SIZE:
        add STRING + NEW_STRING[0] to TABLE
    STRING = NEW_STRING
```