CSC3280 Introduction to Multimedia Systems
Assignment 2 LZW Compression
Due Date: 04/04/2014, 23:59:59

20 marks per day late penalty, fail the course if you copy

I. Introduction

In 1984, Terry Welch proposed the LZW algorithm. The LZW algorithm is based on the LZ77 algorithm, which is proposed by A. Lempel and J. Ziv in 1977. Like LZ77 and LZ78, LZW is a dictionary based compression algorithm and does not perform any analysis on the input text. LZW is widely known for its application in GIF and in the V.42 communication standard. The idea behind the LZW algorithm is simple but there are implementation details that one needs to take care of. In this assignment, you are required to write a variant of the LZW algorithm. The following sections explain the details.

II. Algorithms

• Compression

Figure 1 provides the pseudo-code for the compression procedure:

```
1. Initialize the dictionary, and set P to an empty string
2. While there are still characters to read,
        a. Get the next character, C, from the input text
        b. Search for the string <P, C> in the dictionary
        c. if found,
                i. Remember the code
                ii. Append C to the end of the string P
        d. if not found,
                i. Output the code that corresponds to P
                ii. Add to the dictionary the new entry <P, C>
                iii. P := C
3. Output the code that corresponds to P
```

Figure 1. Procedures for LZW compression

The string P denotes a prefix. During compression, we create a dictionary of strings on the fly. A code corresponds to the location of a string in the dictionary. Assume we are at a particular instance of compression (just after 2.d.iii), a new character is read from the input text. Then, we search for the presence of the new string that is created by appending the character to the prefix P (2.b) in the dictionary. If this string could be found, we will use the new string as the prefix (2.c.i) and repeat the search by appending another new character. Meanwhile, the codeword is memorized (2.c.i).

Conceptually, the iteration is equivalent to reading the longest string that is already contained in the dictionary. When the string could not be found, we will output the codeword for the prefix (2.d.1) and add a new entry to the dictionary (2.d.ii). A practical issue about implementation is how to store the dictionary. Obviously, a linear array suffices, but the performance (i.e. the execution speed) will be unacceptable since the dictionary lookup step (2.b) will take too much time. Alternatives will be to use a tree structure or a hashing function.

- **Decompression**

Figure 2 provides the pseudo-code for the decompression procedure:

```
1. Initialize the dictionary,
2. Get the first code, cW
3. Find cW in dictionary and output the string dictionary(cW)
4. While there are still codes to read,
        a. pW := cW
        b. Get the next code, cW, and find it in the dictionary
        c. if found,
                i. Output the string dictionary(cW)
                ii. Let C be the first character of dictionary(cW)
                iii. P := dictionary(pW)
                iv. Add the string <P, C> to the dictionary
        d. if not found, (exception)
                i. P := dictionary(pW)
                ii. Let C be the first character of P
                iii. Output the string <P, C>
                iv. Add the string <P, C> to the dictionary
```

Figure 2. Procedures for LZW decompression

The decompression process is even easier to understand. It is basically the reverse of the compression process. For a code read from the input stream, the decompression routine will output the corresponding entry in the dictionary (4.c.i) and update the dictionary (4.c.ii, iii, iv). In closer examination, one would find that the decompression process actually *lags* behind the compression process. Hence, there will be the case when a code refers to an entry that has not yet been created. To handle this, we just need to follow the routines (4.d). Besides, cautions must be taken when a new dictionary is created (explained later) in the middle of the decompression process.

NOTE: Assume we are using N-bit codewords, where N>8. Usually, the first 256 entries are reserved for all one-character-long (ASCII) code. Other than the first 256 entries, the last codeword is also reserved. The code 2^N-1 is used to denote "End-of-file".

III. Assignment Details

In this assignment, you are required to implement the LZW algorithm (both the compressor and decompressor) in **ANSI C/C++.** The program should support multiple files (archives).

1. The compressor should be able to compress a file of any length. Also, your program should be able to decompress the compressed file to the original file.

2. When the N-bit code (we use 12 bits in this assignment) word dictionary is full, you should delete the current dictionary and **create** a new dictionary, start with the 256 entries.

3. A source program called "lzw.c" (a source skeleton) is provided. The interface part is completed. Furthermore, we have already provided two routines, *read_code()* and *write_code()* for reading and writing codewords of various lengths. You are required to implement *compress()* and *decompress()* functions in the program. The function *compress()* is used to compress a file using LZW compression while the function *decompress()* is used to decompress the data file. You should put all of your implementations in "lzw.c".

4. Your program should be able to support compression of **multiple files** into one compressed archive. Therefore, you have to save a header in your compressed file with the following format:
<filename1>\n
<filename2>\n
<filename3>\n
…
\n
The header is in stored in plain text, as it will be able to query what is in the file before decompression. Following the header is the compressed files, one after another. When a file is compressed, you have to insert the code "End-of-file" (with value 2^N-1) which indicates a file is ended. While starting to compress a new file, the dictionary is **kept** (if not full) instead of reconstruct a new one.

5. The command line to run your program should have the following format :
For compression
```
> lzw -c <lzw filename> <a list of files>
```
For decompression
```
> lzw -d <lzw filename>
```

6. Your LZW program should be compatible with the sample program, "sample_lzw", provided in the course homepage.

7. You can use any data structures to implement the dictionary so as to speed up the dictionary lookup process. There is bonus (10%) allotted to the speed of execution. Marks will be rewarded to those who have done any optimization.

IV. Marks

Task	Marks
Compression	50
Decompression	50
Performance (bonus)	10

Performance here means the compression speed.

V. Submission Guidelines

1. In all your source files, type your full name and student ID, just like:

```
/*
* CSCI3280 Introduction toMultimedia Systems
*
* --- Declaration ---
*
* I declare that the assignment here submitted is original except for source
* material explicitly acknowledged. I also acknowledge that I am aware of
* University policy and regulations on honesty in academic work, and of the
* disciplinary guidelines and procedures applicable to breaches of such policy
* and regulations, as contained in the website
* http://www.cuhk.edu.hk/policy/academichonesty/
*
* Assignment 2
* Name :
* Student ID :
* Email Addr :
*/
```

Missing of these pieces of essential information will lead to mark deduction.

2. You are **required** to write your programs using **ANSI C/C++** language, since this allows your code to be compatible with compilers on different platforms.

3. Please use zip to pack the source file lzw.c. Name the zip file with your student ID number, e.g s000123.zip.

4. You are required to send your homework to the elearning system.

https://elearn.cuhk.edu.hk/

5. 20 Marks will be deduced per day delayed *including public holidays & Sundays*. You will fail the course if you copy others' work.