CSCI 3280 Introduction to Multimedia Systems
Spring 2017, Assignment 2 - LZW Compression

Due Date : Mar. 31, 2017 (11:59pm)
Submission via elearn.cuhk.edu.hk
Late submission penalty : 10-point deduction per day
PLAGIARISM Penalty : Whole Course Failed

1. 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 should be taken care of. In this assignment, you are required to write a variant of the LZW algorithm. The following sections explain the details.

2. Algorithm for Compression

```
CodeDict     : A dictionary which maps a given char. or sequence of char. to a CODE.
P            : Prefix string, to store a sequence of characters
C            : The latest SINGLE character read from the input text file
X            : A matched code to be output to Archive

1. Initialize the dictionary CodeDict with the first 256 ASCII codes (i.e. 0-255).
2. Let P = NULL.

3. While there are still characters to be read from input file:
     A. Get a new character C from the input file.
     B. Search for the string <P,C> in CodeDict:
          i. if FOUND:
                 Let X = CodeDict[<P,C>]
                 Let P = <P,C>
         ii. if NOT FOUND:
                 Output the code X that corresponds to P
                 Add a new entry <P,C> to CodeDict
                 Let P = C

4. Output the code that corresponds to P
```

The string **P** denotes a prefix. During compression, we create a dictionary **CodeDict** of strings on the fly. For each entry of the dictionary, there is a corresponding code to the string (an arbitrary example: string "abc" has a code of 280). Assume we are at a particular instant of compression, a new character **C** is read from the input file. Then, we search in the dictionary for the presence of the new string **<P,C>** that is created by appending the character **C** to the prefix **P** (**step 3.B**). If this string **<P,C>** could be found (**step 3.B.i**), the code found is memorized as **X**, and we will use the string **<P,C>** as a new prefix. The search is then repeated by appending another new character from the input file.

Conceptually, the iteration is equivalent to reading the longest string that is already contained in the dictionary. When the string could not be found (**step 3.B.ii**), we will output the memorized codeword **X** of its prefix and add a new entry to the dictionary for this string.

A practical issue about the implementation is how to store the dictionary which allows efficient lookup. Obviously, a linear array suffices, but the performance (i.e. the execution speed) will be unacceptable since the dictionary lookup step (**step 3.B**) will take too much time. Alternatives will be to use a tree structure, hashing function or some clever data structure you may design.

3. Algorithm for Decompression

```
StrDict      : A dictionary which maps a given CODE to a char. or sequence of char.
PW           : Previous code read from the archive
CW           : Current code read from the archive
S            : String to be output to output file


1. Initialize the dictionary StrDict with the first 256 ASCII codes (i.e. 0-255).
2. Let PW = first code read from the archive.
3. Let C = StrDict[PW] and output C.


4. While there are still codes to be read from the archive:
      A. Get a new code CW from the archive.
      B. Search for the code CW in StrDict:
            i. if FOUND:
                  Let C = First character of StrDict[CW]
                  Let S = StrDict[CW]
            ii. if NOT FOUND:
                  Let C = First character of StrDict[PW]
                  Let S = <StrDict[PW],C>
      C. Output S
      D. Let P = StrDict[PW]
      F. Add a new entry <P,C> to StrDict
      G. Let PW = CW.
```

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 and update the dictionary. 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 (**step 4.B.ii**)..

NOTE: We will be using 12-bit codewords. The first 256 entries are reserved for all one-character-long (ASCII) code. Apart from the first 256 entries, the last code is also reserved. The code 4095 is used to denote "End-of-file".

4. 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 be able to archive multiple files

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 12-bit code or string dictionary is full, you should delete the current dictionary and create a new dictionary, start with the 256 entries.

3. A source skeleton program called **"lzw.c"** is provided. The complete I/O interface is complete. We have already provided two functions **readCode()** and **writeCode()** for reading and writing N-bit codewords of various lengths, you will use 12-bit. 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**
> **…**
> **<filenameN>\n**
> **\n**

Two convenience functions namely **readArchiveHeader()** and **writeArchiveHeader()** are also included. This 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 4095) which indicates a file is ended. While starting to compress a new file, the current 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 :

> **Compression:**
> **> lzw –c <lzw filename> <a list of files>**

> **Decompression:**
> **> lzw –d <lzw filename>**

6. 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.

7. The compression ratio of your program must be reasonable. For example, if you try to compress a text file, it may achieve compression ratio around 2:1, and if you try a map.jpg the compressed file may be larger than map.jpg, since the jpg is already compressed and your algorithm may be worse than .jpg for image files.

5. Marking Scheme

Task	Marks
Compression code correctness	**30**
Compression code documentation (code comments)	**20**
Decompression code correctness	**30**
Decompression code documentation (code comments)	**20**
Data Structure Performance (bonus)	**10**

6. Submission Guidelines

1. In all your source files, type your full name and student ID as follows:

```
/*
* CSCI3280 Introduction to Multimedia 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 :
*/
```

2. You are required to complete **"lzw.c"** using ANSI C/C++ language, you are required to **submit source code only**. We will use Visual Studio 2015 C++ compiler and have your program compiled via **visual studio command prompt** with the following command line. *__Please make sure your source code gets compiled well with it, 'failed to compile' receives 10-point deduction for each failed source__*.

<div align="center">

C:\> cl.exe lzw.c

</div>

3. Please use zip to pack the source file **lzw.c**. Name the zip file with your student ID number, e.g **s11550XXXXX.zip**.

4. You are required to submit your work via the elearning system: https://elearn.cuhk.edu.hk/

5. 10 points deduction per day will be applied to all late submission, and you will fail the course if you copy others' work.