CSCI 3280 Introduction to Multimedia Systems

Spring 2023, Assignment 2 – LZW Compression

Due Date: Mar. 16th, 2023 (11:59pm) Submission via Blackboard
Late submission penalty: 10% deduction per day (maximum 30%)
PLAGIARISM penalty: Whole Course Failed

Introduction

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. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. In this assignment, you are required to write an LZW algorithm (both the compression and decompression). The following sections explain the details.

Code Dictionary

Code dictionary is a dictionary holding the correspondence between strings and codes. By using the LZW algorithm, we can construct the dictionary on the run, and represent a string with its corresponding code in the dictionary so that less number of bits can be used to store the same information.

In this assignment, we will be using **12-bit** code sequence to represent strings of 8-bit characters in the original file. Because a 12 bit can have values from 0 to 4096, the dictionary can hold 4095 different strings (except code 4095 which reserved as EOF code). The first 256 entries of the dictionary need to be initialized as the 256 ASCII characters to represent all single-character strings.

An interesting aspect of this algorithm is that we don't need to store the dictionary in the compressed file; instead, we discard the dictionary once the files are compressed, and restore it from the compressed data when we start to decompress.

Once you dictionary is full, you need to delete the dictionary and start a new dictionary.

You can refer to the example below. The first 256 and the last entries have predefined values, while the remaining entries can be used to hold multi-character strings encountered during the compression.

Example Code Table

	code number	translation
identical code	0000	0
	0001	1
	⋮	⋮
	0254	254
	0255	255
unique code	0256	145 201 4
	0257	243 245
	⋮	⋮
	4095	xxx xxx xxx

Algorithm for Compression

You can follow the procedure below to compress data using LZW algorithm.

1) Initialize **DICT** with the first 256 entries
2) Define 2 variables: **STRING, CHAR**
3) Get first byte from file and store in **STRING**
4) While there is more byte in file, read 1 byte and store in **CHAR**
5) If **(STRING+CHAR)** is in **DICT**, let **STRING = (STRING+CHAR)**
6) Else, output **code of STRING**, add **(STRING+CHAR)** to **DICT** and let **STRING = CHAR**
7) Output code of **STRING**

In this way, we will be scanning through input character stream, and constantly adding incoming characters into a variable **STRING**. Once the **STRING** + its following character **CHAR** generate an unseen new string, we will record this **STRING+CHAR** into the **DICT**, output the code corresponding to **STRING**, and set **STRING** to be the new **CHAR** (we will not be using the newly added code to represent the first occurrence of this unseen combination, but to use it when we see this combination again).

Make sure you start a new dictionary when the current one has no new entries. On the other hand, you should **not** start new dictionaries when you finish compressing one file and start with another, unless the dictionary is full.

Algorithm for Decompression

1) Initialize **DICT** with the first 256 entries
2) Define 4 variables: **OCODE, NCODE, STRING, CHAR**
3) Get first code from file and store in **OCODE**
4) While there is more code in file, read 1 code and store in **NCODE**
5) Let **STRING =** translation of **OCODE**, and output **STRING**
6) If entry **NCODE** is non-empty in **DICT**, let **CHAR =** first char of translation of **NCODE**
7) Else, let **CHAR =** first char of **STRING**
8) Add **(STRING+CHAR)** to **DICT** and let **OCODE = NCODE**
9) Output translation of **NCODE**

Following this procedure, we scan through the compressed code sequence, each time load a code into **OCODE** and the following one into **NCODE**.

Remember that we output the **code of STRING** and save **(STRING+CHAR)** into the **DICT** if and only if the current **STRING** is interrupted by the incoming **CHAR** during the compression. To do the inverse process of the Compression, for each **OCODE, NCODE** pair in the sequence, we need to 1) decode **OCODE** back to **STRING** and write **STRING** to the decompressed file, and 2) save **(STRING +** the following **CHAR)** into **DICT**.

Because **NCODE** is representing the string following the one represented by **OCODE**, in most cases we can use the first character of the translation of **NCODE** as **CHAR**. But the tricky part is that sometimes **NCODE** does not exist in **DICT**, waiting for the newest **(STRING+CHAR)** to be inserted first to know its value. In this case, we can reason that **CHAR==(STRING+CHAR)[0]**, and **CHAR==STRING[0]** for any **STRING** that has length >= 1. As a result, we can instead use **STRING[0]** as the value of **CHAR**.

If you implemented the algorithm correctly, the **DICT** in decompression should be exactly the same with the **DICT** in compression. Similar to the compression part, you need to start new dictionaries on demand.

Basic Part (80%)

1) Write a **C/C++** program based on the given skeleton code to perform compression and decompression of files using LZW.

2) Program has to support compressing **multiple files** into one file and decompress back to multiple files. The code sequences of different files should be separated by a **<EOF>** code. There is **no** need to extract filenames from input file paths; you can simply pass the input file paths to the writefileheader function. An example of compressed file (the header is in plain text) is given as:

C:\File1.txt\n
C:\File2.txt\n
\n
<Code sequence for File1.txt> <EOF(4095)> <Code sequence for File2.txt> <EOF(4095)>

3) The code size has to be **12-bit** (dictionary size = 4096).

4) The decompressed file has to be **exactly the same** with the input file.

5) Program will be tested with **both ASCII files and binary files**. (You program may fail on binary files if it cannot handle negative byte values properly)

6) The input file can have arbitrary length. (You should ensure your program to work well with several mega-bytes of input data)

7) The compressed file will **not** be compared with a sample answer, but there will be penalty if your compressed file is more than 10% larger than the expected compressed size.

8) The program should run in **Windows** command prompt as a console program and accepts the following arguments:

For compression:

lzw –c <compressed file name> <file 1> <file 2> …

(Example) **lzw –c C:\output.lzw C:\text.txt C:\image.png**

For decompression:

lzw –d <compressed file name>

(Example) **lzw –d C:\output.lzw**

Enhancement Part (20%)

In this part, you can implement features related to LZW. Here are some examples:

- The LZW algorithm we implemented in the basic part use arrays as the code dictionary, which is quite slow to lookup in practice. You can implement **alternative data structures or algorithm** (like trees or hashing) for better performance of the dictionary. Analyzing the time and space complexity of your advanced dictionary implementation with the one in the basic part gains more marks.
- There are some **image formats** like GIF and TIFF that use LZW compression to compress their data, you can write a program to compress an image into such file formats.

You can also implement other interesting features about LZW compression.

Please write a report to describe your enhancement features. For each additional feature:

1) The explanation of the features
2) The source code segment related to this feature
3) Sample runs (including the execution code and output) (if applicable)
4) The techniques used (if applicable)
5) References (E.g., the feature you implemented is based on an algorithm found on Internet or a reference book)

We will do the grading and test the correctness of the source code based on the features you mentioned in the report. You are required to provide sufficient information in details to facilitate the grading. No marks will be given if you implemented a feature in the source code without mentioning it in the report.

If the features claimed in the report can only be barely used (e.g., with a lot of bugs) or even do not exist in the source code, marks will be deducted.

General Specifications

1. Program should be coded in **ANSI C/C++** and uses libraries listed in the skeleton code only.
2. No additional libraries are allowed except for the enhancement part.
3. The following files are provided for you to write and test your program.
    a. **`lzw.cpp`**: The skeleton code for LZW program.
        i. **`readfileheader, writefileheader`** functions for helping you to read and write file headers in the required format.
        ii. **`read_code, write_code`** functions for reading and writing N-bit code from and to files.
    b. Three test files: **`Windows.txt, CSE.txt, lena.bmp`**
    c. **`lzw.exe & compressed.lzw`**: A sample program for compression and decompression, and a sample compressed file containing all three test files.
    (You don't need to have exact same output with the sample program)
4. You are required to submit source codes only. We will use Visual Studio 2019 C++ compiler and have your program compiled via visual studio command prompt and the following command line. A second chance to compile with g++ on Linux will be given if it fails with cl in the first attempt.
    - **`C:\> cl lzw.cpp`**
    - **`C:\> cl lzw_enhancement.cpp`**

    `lzw.cpp` is the provided skeleton code for the basic part of the assignment. For the enhancement part, copy the source file and name it **`lzw_enhancement.cpp`**; it will be compiled and executed in the exact same way.

Please make sure that your source code gets compiled well, failing to compile in both attempts will receives 10-point deduction for each failed source.

Submission (Deadline: Mar. 16th, 2023 11:59pm)

We expect the following files zipped into a file named by your **student ID** (e.g., 1155xxxxxx.zip) and have it uploaded to the course's Blackboard system.

1. **`report.pdf`** (Tell us anything that we should pay attention to, especially about the enhancement part)
2. **`lzw.cpp`** (Basic requirement source code)
3. **`lzw_enhancement.cpp`** (Basic + enhancement part source code)
4. 3rd party libraries you used in your enhancement part. Separate them from your own files, and state in you report which file is written by yourself.

Other files that are provided are not required to be submitted.

*******IMPORTANT*******

All code in the two listed source files should be implemented by yourself. Any kind of plagiarism (including copying online source code or/and code of classmates) in all assignment parts (both the general part and the enhancement part) will not be tolerated and will be subjected to disciplinary penalties.