CS 1501 Project 2

Released: Monday, September 30

Due: Sunday, October 13, 11:59 PM

Goal

To understand the innerworkings and implementation of the LZW compression algorithm, and to gain a better understanding of the performance it offers.

Background

As we discussed in lecture, LZW is a compression algorithm that was created in 1984 by Abraham Lempel, Jacob Ziv, and Terry Welch. In its most basic form, it will output a compressed file as a series of fixed-length codewords. This is the approach implemented in the LZW code provided by the authors of the textbook. As we discussed in class, *variable-width* codewords can be used to increase the size of codewords output as the dictionary fills up. Further, once the dictionary fills up, the algorithm can either stop adding patterns and continue compression with only the patterns already discovered, or the algorithm can reset the codebook to find new patterns. The LZW code provided by the textbook authors simply continues to use patterns added to the codebook.

For this project, you will be modifying the LZW source code provided by the authors of the text book to use variable-width codewords, and to optionally reset the codebook under certain conditions. With these changes in hand, you will then compare the performance of your modified LZW code with the provided LZW code, and further with the performance of a widely used compression application of your choice.

Specifications

1. Download copies of the 14 example files from here.
    - Do not upload these files to your Box folder!
2. Make a copy of `LZW.java` named `MyLZW.java`. You will be modifying this file for your assignment. Note that `LZW.java` is the example LZW code provided by the textbook.

3. Before making the required changes to `MyLZW.java`, you will need to read through the code, and run example compressions/expansions to understand how it is currently working. Note that `LZW.java` (and hence your `MyLZW.java`) requires the following library files (also developed by the textbook authors): `BinaryStdIn.java`, `BinaryStdOut.java`, `TST.java`, `Queue.java`, `StdIn.java`, and `StdOut.java`. These files have already been added to your repository.

4. With a firm understanding of the provided code in hand, you can proceed to make the following changes to `MyLZW.java`:

    - Make it so that the algorithm will vary the size of the output/input codewords from 9 to 16 bits.
    - The codeword size should be increased when all of the codewords of a previous size have been used
    - Modify the code to have three options when the codebook is filled up (i.e., when all 16-bit codewords have been used):

        1. **Do Nothing mode** Do nothing and continue to use the full codebook (this is the mode implemented by `LZW.java`).
        2. **Reset mode** Reset the dictionary back to its initial state so that new codewords can be added. Be careful to reset at the appropriate place for both compression and expansion, so that the algorithms remain in sync. This is very tricky and may require alot of planning in order to get it working correctly.
        3. **Monitor mode** Initially do nothing (keep using the full codebook) but begin monitoring the *compression ratio* whenever you fill the codebook. Define the compression ratio to be the size of the uncompressed data that has been processed/generated so far divided by the size of the compressed data generated/processed so far (for compression/expansion, respectively). If the compression ratio degrades by more than a set threshold from the point when the last codeword was added, then reset the dictionary back to its initial state. To determine the threshold for resetting you will take a ratio of compression ratios [(old ratio)/(new ratio)], where old ratio is the ratio recorded when your program last filled the codebook, and new ratio is the current compression ratio. If the "ratio of ratios" exceeds 1.1, then you should reset.

            For example, if the compression ratio when you start monitoring is 2.5 and the compression ratio at some later point is 2.3, the ratio of ratios at that point would be 2.5/2.3 = 1.087, so you should not reset the dictionary. Continuing, if your compression ratio drops to 2.2, the ratio of ratios would become 2.5/2.2 or 1.136. This means that your ratio of ratios has exceeded the threshold of 1.1 and you should now reset the dictionary. Be very careful to coordinate the code for both compression and expansion so that it works correctly.

    - You cannot encode either a codeword size switch or a codebook reset into the compressed file using

a delimiter. You must have your compression/expansion code detect when to switch codeword sizes and reset based only on the state of the codebook.

- The mode to be used should be chosen by the program during compression. Whichever mode is used to compress a file should also be used to expand the file. However, you should not require the user to state the mode to use for expansion. The mode used to compress a file should be stored at the beginning of the output file, so that it can be automatically retrieved during expansion. To establish the mode to be used during compression, your program should accept 3 new command line arguments:
  - `n` for Do Nothing mode
  - `r` for Reset mode
  - `m` for Monitor mode
- Note that the provided LZW code already accepts a command line argument to determine whether compression or expansion should be performed (`-` and `+` , respectively), and that input/output files are provided via standard I/O redirection (`<` to indicate an input file and `>` to indicate an output file). Hence, your new arguments should be handled *in addition to* what is provided. For example, to compress the file `foo.txt` to generate `foo.lzw` using Reset mode, you should be able to run:

  ```
  java MyLZW - r < foo.txt > foo.lzw
  ```

  Similarly, to expand `foo.lzw` into `foo2.txt` , you should run:

  ```
  java MyLZW + < foo.lzw > foo2.txt
  ```

  Note that this example does not overwrite `foo.txt` . This is a good approach to take in testing your programs so that you can compare `foo.txt` and `foo2.txt` to ensure that they are the same file.

5. Once all of the required changes have been made to `MyLZW.java` , you should evaluate its performance on the 14 provided example files: `all.tar` , `assig2.doc` , `bmps.tar` , `code.txt` , `code2.txt` , `edit.exe` , `frosty.jpg` , `gone_fishin.bmp` , `large.txt` , `Lego-big.gif` , `medium.txt` , `texts.tar` , `wacky.bmp` , and `winnt256.bmp` . Specifically, for each of the provided example files, measure the original file size, compressed file size, and compression ratio (original file size / compressed file size) when compressed using the following techniques:
   - The unmodified `LZW.java` program (i.e., 12-bit codewords)

- Your `MyLZW.java` (variable width codewords) using Do Nothing mode
- Your `MyLZW.java` (variable width codewords) using Reset mode
- Your `MyLZW.java` (variable width codewords) using Monitor mode
- Another existing compression application of your choice (e.g., 7zip, WinZIP, gzip, bzip2). You should organize your results of these compressions/expansions into a table in a text file named `results.txt` and submit it along with your code.

Submission Guidelines

- Upload your submission to the provided Box folder named `cs1501-p2-abc123`, where `abc123` is your Pitt username.
- **DO NOT** upload the example files to your Box folder.
- **DO NOT** upload any IDE package files.
- You must name the primary driver for your program `MyLZW.java`, and it must be outside of any package.
- You must be able to compile your program by running `javac MyLZW.java`.
- You must be able to run your program as shown in the above examples.
- You must fill out `info_sheet.txt`.
- The project is due at 11:59 PM on Sunday, October 13. Upload your progress to Box frequently, even far in advance of this deadline. **No late assignments will be accepted.** At the deadline, your Box folder will automatically be changed to read-only, and no more changes will be accepted. Whatever is present in your Box folder at that time will be considered your submission for this assignment—no other submissions will be considered.

Additional Notes and Hints

- In the authors' code, the bits per codeword (`W`) and number of codewords (`L`) values are constants. In `MyLZW`, you will need them to be variables. As the bits per codeword value increases, so does the number of codewords value.
- The TST the authors use can grow dynamically, so it does not matter how large the dictionary will be. However, for the `expand()` method, an array of String (`String[]`) is used for the dictionary. Make sure this is large enough to accommodate the maximum possible number of codewords!
- Carefully trace what your code is doing as you modify it. You only have to write a few lines of code for this program, but it could still require a substantial amount of time to get to work properly. The trickiest parts occur when the bits per codeword values are increased and when the dictionary is reset. I recommend

tracing these portions of code, either on paper or with output statements, to make sure your compression and expansion sections are treating them correctly. One idea is the have an extra output file for each of the `compress()` and `expand()` methods to output any trace code. Printing out (codeword, string) pairs in the iterations just before and after a bit change or reset is done can help you a lot to synchronize your code properly.

- Be especially careful with the dictionary reset and monitor compression ratio options. These are very tricky and take alot of thought to get to work. Think about what happens when the dictionary is reset and what is necessary to do in the `compress()` and `expand()` methods. I recommend getting the variable width codeword part of the program to work first, and then moving on to implementing Reset mode and Monitor mode.

- Start on this project early! Not only will the implementation be tricky, but you will need to finish the programming portion of your project with enough time left over to gather results using your code to compress the example files.

- Note that `LZW.java` (and consequently your `MyLZW.java`) rely on redirecting standard in and standard out to the input and output files (respectively). An overview of I/O redirection can be found here. Note that a consequence of this is that any text printed to standard out (i.e., via `System.out.println()`) will be redirected to the output file instead of the terminal. Standard error, however, should still be displayed to the terminal, and hence, you can use `System.err.println()` to output debugging information. This I/O redirection may also complicate running `MyLZW` from some IDEs. If you are having trouble running `MyLZW` from your IDE, please try to run your program from the command line.

- Consider the notes in `LZW.java` (and `TST.java`) concerning the speed of the `substring()` function. In order to run your experiments more quickly, you may want to edit `MyLZW.java` and `TST.java` to remove all calls to `substring()`. There is no penalty for continuing to use `substring()` for this assignment, but you will experience noticeably slow performance.

Grading Rubric

Feature	Points
Command-line arguments are interpreted as specified	10
Variable-width keywords (9–16 bits) working properly	25
Reset mode implemented and working properly	20
Monitor mode implemented and working properly	20

Experimental results	Points
Unmodified `LZW.java`	4
Variable-width codewords (`MyLZW.java`) with Do Nothing mode	4
Variable-width codewords (`MyLZW.java`) with Reset mode	4
Variable-width codewords (`MyLZW.java`) with Monitor mode	4
An appropriate popular compression application	4

Other	Points
Assignment info sheet/submission	5