CS 1501 Programming Project 3

Lempel-Ziv-Welch Compression

Online: Friday, June 16, 2017
Due: All materials zipped into a single .zip file posted to the course submission site by 11:59PM on **Friday, June 30, 2017**
Late: All materials submitted by 11:59PM on Monday, July 3, 2017.

Purpose: The purpose of this assignment is for you to fully understand the LZW compression algorithm, its performance and its implementation.

Procedure:

1) Thoroughly read the description/explanation of the LZW compression algorithm as discussed in lecture and in the Sedgewick text.
2) Read over and make sure you understand the code provided by Sedgewick in file `LZW.java`.
3) There are two fundamental problems with this code as given:

   a) The code reads the entire file as a single string, then compresses the characters in this string. This is problematic in that a very large file could not effectively be fit into a single Java string. Further, since JDK 7, the substring() method in the Java String class actually generates a new String object, causing a lot of overhead when used in the manner shown in the author's LZW.java file. In fact, with the textbook code using JDK 7+ and a large input file, the compression can take an excessive amount of time (ex: hours). This is due primarily to the code on line 30 of the LZW.java file:
   ```
   input = input.substring(t);
   ```
   The purpose of this statement is to shift down in the string past the characters that have already matched in the LZW dictionary. However, since in JDK 7+ the substring() method generates a new String, the effect of this statement is to create a copy of the entire file string, minus a few characters at the beginning that already matched in the dictionary. It is easy to see how this process repeated many many times (thousands and even millions) can slow the program execution incredibly.
   b) The code uses a fixed length, relatively small codeword size (12 bits). With this limit, the program will run out of codewords relatively quickly and will not handle large files (especially archives) well.

4) In this assignment you will modify the author's code so as to correct these two problems. Proceed in the following way:
   a) Download the implementation of the algorithm provided and get it to work (you will need to also download several other files to get it to compile – they are on the Handouts page). Follow the instructions in the comments and run the program on a few test files to get familiar with using it. Try running the program with a large input file to see the behavior discussed above.
   b) Examine the code very carefully, convincing yourself exactly what is accomplished by each function and by each statement within each function.
   c) Copy the code to a new file called LZWmod.java and modify the code so that during compress() the input file is read as a stream of characters / bytes rather than as a single string. There are many ways to do this and most of the details are up to you. However, here are a few requirements:
      i) The input file must be read in a single character / byte at a time. Look over some classes in java.io that might be appropriate to use in this case. You may also utilize any of the author's IO classes if you wish. The idea is that rather than using the longestPrefixOf() method on a single String, you will find the longest prefix yourself by repeatedly reading and appending characters and looking the prefix up in the symbol table.
      ii) The "strings" that are looked up in the dictionary must actually be StringBuilder objects. Using StringBuilder rather than String will allow the values to be updated more efficiently (ex: appending a character onto the end of the StringBuilder will not require a new StringBuilder object to be created, as it would with a String).

iii) The dictionary in which the StringBuilders are looked up must be some type of symbol table where the keys are StringBuilder objects and the values are arbitrary Java types. The dictionary must have (average) constant lookup time. Note that the predefined Java Hashtable and HashMap classes are not appropriate in this case because StringBuilder does not override the hashCode() method to actually hash the string value. Some options that could work include modifying one of the author's hash classes or the author's TrieST class so that they work for StringBuilder. Another option is to modify your DLB class from Assignment 1 to allow it to store the codewords.

Note that you should not have to modify the expand() code for this feature at all. Also note that this modification will take just a few lines of code but it may take a lot of trial and error before you get it working properly. **I strongly recommend getting this to work before moving on to part d) below.** However, if you cannot get this to work, you can still get credit for part d) below by performing it on the original LZW.java file.

d) Also modify the code so that the LZW algorithm has a varying number of bits, as discussed in lecture. Your codeword size should vary from 9 bits to 16 bits, and should increment the bitcount when all codes for the previous size have been used. This also does not require a lot of modification to the program, but you must REALLY understand exactly what the program is doing at each step in order to do this successfully. Once you get the program to work, thoroughly test it to make sure it is correct. If the algorithm is correct, the byte count of the original file and the uncompressed copy should be identical. Some hints about the variable-length codeword implementation are given later on in this assignment.

5) To prevent headaches (especially during debugging), when testing your program you do should not replace the original file with the new one (i.e. leave the original file unchanged) when compressing. Thus, make sure you use a name for the output file that is different from the input file. For example, to compress file bogus.txt you may do the following:

```
$ java LZWmod - < bogus.txt > bogus.lzw
```

If you then want to decompress the bogus.lzw file, you might enter at the prompt

```
$ java LZWmod + < bogus.lzw > bogus2.txt
```

The file bogus2.txt should now be identical to the file bogus.txt.

6) Once you have your variable code length program working, you should analyze its performance. I will provide you with a number of files to use for testing – see the Assignments page for the link. Specifically, you will compare the performance of 3 different implementations:

a. The original `LZW.java` program using codewords of 12 bits (i.e. the way it is originally – you don't have to change anything)

b. Your modified `LZWmod.java` program with the streaming input text and variable length BITS from 9 to 16 as explained above

c. The predefined Unix `compress` program (which also uses the lzw algorithm). If you have a Mac or Linux machine you can run this version directly on your computer. If you have a Windows machine, you can download this version of compress.exe (obtained from www.willus.com/archive/unixcmds.zip). To decompress with this program use the flag "-d".

Run all programs on all of the files and for each file record the **original size**, **compressed size**, and **compression ratio** (original size / compressed size). In addition to the files in the directory, also test copies of your **source code** and of your **.class files** for this project.

[Note: Because of the aforementioned run-time issues with the author's original code, it may take a prohibitive amount of time to get results for the larger files. However, it should eventually complete – just leave yourself a lot of time for your runs.

7) Write a short (~2-3 pages) paper that discusses each of the following:

a) What your modifications were to the program and how you got the streaming character input and variable length codes to work. Explain in detail everything you did to the program and why, especially issues that caused you any grief.

b) How all three of the lzw variation programs compared to each other (via their compression ratios) for each of the different files. Where there was a difference between them, be sure to explain (or speculate) why. To support your assertions, **include a table showing all of the results of your tests** (original sizes, compressed sizes and compression ratios).

   c) For all algorithms, indicate which of the test files gave the best and worst compression ratios, and speculate as to why this was the case. If any files did not compress at all or compressed very poorly (or even expanded), speculate as to why.

8) Submit a single .zip file containing your **modified LZWmod.java source code and any other source files necessary to compile your code.** Also include in your .zip file your paper along with your Assignment Information Sheet. **NOTE: DO NOT submit any of the test files (input or output) – this will waste an incredible amount of space on the submission site!**

9) **W Section**: No extra paper is required. However, your required paper (from 7) above) should be 4-5 pages and it will be evaluated for both form and content, and will be weighted more heavily than the papers of students in the non-W sections. It should go into detail on all of the issues mentioned in 7) above, especially the analysis in parts b) and c). **Be sure to submit your paper as a .doc or .docx file so that we can easily insert comments / changes.** You may also be required to write a revision of your paper later on.

10) Hints:
   a) In the author's code the bits per codeword (W) and number of codewords (L) values are constants. However, in your version you will need them to be variables. Clearly, as the bits per codeword value increases, so does the number of codewords value.
   b) The symbol table that you use for the compression dictionary (ex: Trie, DLB) can grow dynamically, so you do not have to alter this as the codeword size increases. However, for the expand() method an array of String is used for the dictionary. Make sure this is large enough to accommodate the maximum possible number of codewords.
   c) 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. Clearly the trickiest parts occur when the bits per codeword values are increased. I recommend tracing these portions of code, either on paper or with output statements to make sure your compress and expand 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.