Programming Assignment 3

Compression

Due date: Mar. 02 by 11:00 pm

Overview

For this homework assignment we will implement and experiment with methods for data compression, using two lossless data compression algorithms: Huffman coding and LZW compression.

Objectives

- Apply core data structures and algorithms from the course, such as priority queues, Huffman trees, and tries to a real-world problem.
- Gain an understanding of major data compression algorithms, including Huffman coding and LZW compression.
- See why good algorithms matter, not only for good results (in compressing data) but also for getting practical performance.
- Gain experience working with more realistic problems that are larger and open-ended.

Starter Code

Download the starter code for assignment (zip file). Once you finish implementation, submit java files to Autolab.

What to Submit

You FTP the following java files

- HuffmanDecode.java
- HuffmanEncode.java
- DictionaryTrie.java
- LZW.java

Time Management

Due to the size of this assignment, it is normal to feel somewhat overwhelmed. Make sure to get an early start and think before you hack! Breaking it up will allow you to treat this homework assignment essentially as two smaller assignments, and also give you the flexibility to work on one part while you're still debugging another part. Don't wait until the last minute to start on all the parts!

Priority Queue

You use the Java implementation of the priority queue.

The Huffman Algorithm (30 points)

Huffman's part involves the implementation of plain Huffman compression as discussed in lecture. You are to implement `HuffmanEncode.java` and `HuffmanDecode.java`.

Let us consider the process of compression. First it reads in the file and creates a list of frequencies for each byte (aka char). It passes these frequencies to the `HuffmanTree` method that takes a `Map`. Then the compressor calls the `writeHeader` method which emits the tree recursively, using `writer.writeByte(byte)` to write each data item. Next, it write the input file size. Finally, the compressor reads each character in the file and writes the codeword by using `encode`

The header is written as follows. You should use a recursive helper method. To write the header, you start by looking at the root. If the node you are looking at is a parent node (i.e. is not a leaf) you output the bit PARENT (a static constant) and then call the method recursively on the node's left child, and then on the node's right child. If the node you are looking at is a leaf node, you output the bit LEAF (another static constant) and then output the character (aka byte) at that node. You should be able to figure out how to read the header and use it to rebuild the tree by yourself. We specify this header format so that every implementation generates a header of the same size.

Now we consider decompression. First it reads the header from the file using the `HuffmanTree` method that takes a `BitReader`. The method reads in the header, calling the `readByte` method of the `BitReader` whenever it sees a leaf in the file header. Then it reads in each Huffman code and outputs the symbol.

The Lempel-Ziv-Welch Algorithm (20 points)

The second compression algorithm of this assignment involves implementing a more sophisticated method for data compression, based on a version of the Lempel-Ziv-Welch algorithm. The method automatically builds a dictionary of previously seen strings in the text being compressed. Unlike to the Huffman algorithm, the dictionary does not have to be transmitted with the compressed text.

The dictionary starts with 256 ASCII entries, one for each possible character. Every time a string is not in the dictionary, a new entry in the dictionary is

created. Each entry in the initial dictionary is 16 bits long. **We will not test** it with any input that gives codewords longer than 16 bits. As the dictionary grows, its size increases to the maximum value. You throw an `IllegalArgumentException` when it exceeds the limit..

It's recommended that you implement compression with a trie. Every character you read in represents taking one step down the trie; when you "fall off" the trie, you output a codeword and add a new one to the dictionary. You should use a HashMap in the trie nodes to store the children, since having a 256-element array will be a big waste of memory. Feel free to reuse the trie you implemented in the previous assignment, however your trie the trie needs to be able to handle all 256 ASCII characters. **We will not test your trie.**

When decompressing, just use a HashMap to store the dictionary. Note, you must handle the degenerate case of decompression - when a codeword you read in is not in the dictionary. There is a specific way to deal with this situation.

Notes

The keyword char should never appear in this assignment, anywhere, ever. These classes deal with bytes and ints, not chars. With chars, all sorts of issues related to encoding arise. Using the raw bytes bypasses these issues.

Call writer.flush() at the end of both LZW methods! If you don't, you might get corrupted output data.

Testing:

Writing tests is good! By now you should realize that testing is highly beneficial. Be sure to test major compress/decompress functionality, as well as to test individual parts like building the HuffmanTree.

This is a good time to point out that good code coverage does not neccesarily equal good grades. The few hand cases, if written correctly can achieve full code coverage of the compression stage. They are, however, insufficient to thoroughly test your code.

Victor S. Adamchik, CMU, 2016