JET Audio Encoder Final Report

Authors: Taylor Autrey

Eric Addison

Josh Musick

**Introduction:**

In a world of analog to digital conversion, there are many questions that can be asked regarding how to do it effectively and efficiently. This area of exploration took major root as a result of two things. First, an internet connection became common in many households, and secondly, the desire / need to transfer audio files through the internet. When it started, the raw audio signal that was captured on cd quality files was far too large to transmit economically. Therefore, the need to compress the audio files into more manageable file sizes became necessary.

When compressing data, there are two types of compression that can be performed, loss-less and lossy. The names give obvious clues about them, but in general a loss-less compression scheme allows the original file to be completely reconstructed from the compressed file, while lossy schemes allow for “good” reconstruction of the original file. The measure of how good a compression algorithm is can be determined by a few factors, the compressed file size, the time it takes to make that compressed file, and the quality of the resulting file. In this project we embark to identify what specific parts of a compression scheme gives those three qualities.

Compression of normal data files dictates that any compression scheme should be loss-less, but audio files do not have that stringent of a requirement. Therefore, some liberty can be taken in how that process happens. Because audio signals for common audio files are actually the overlapping of many frequency bands at a given time, there became some obvious gains in simplifying the audio signal that is eventually saved. The first gain is the removal of frequencies that are actually overpowered by stronger frequencies at a given point in time. There are also some frequency bands which are outside the spectrum of human hearing. By exploiting these two areas, many possible techniques were developed. The original intent of this project was to analyze multiple compression schemes, but due to very specific encoding standards, the specific algorithms involved in the compression of MP3s are what we focused on. There turned out to be many different algorithms involved with interesting properties surfacing with specific input types.

In this project, we have implemented a pseudo-mp3 encoding program. The only part our program does not do is the formatting of an actual mp3 file, due to the highly detailed format of an mp3 file, which is outside the scope of this project. We took the major pieces of the encoding pipeline and implemented them with some variable controls to analyze the performance of the algorithms with a variety of inputs. The major pieces of the pipeline include: File I/O, a filterbank, a Modified Discrete Cosine Transform (MDCT), byte bufferizers, Huffman encoding, and serializing out the compressed file. The program is able to take a wav file, compress it, and then decompress the compressed file. This allows us to evaluate the “quality” of the compression, or lack thereof.

**Filterbank**

Cool stuff about what the filterbank does…

**Modified Discrete Cosine Transform (MDCT)**

Cool stuff about what the MDCT does…

**Byte Bufferizers**

Cool stuff about what the byte bufferizer does…

**Huffman Encoder**

Recall that the strategy for Huffman encoding is to build a frequency table of characters in a file and then use a variable-length code to represent the characters, with the more frequent characters having shorter codes.  In our implementation, the characters are bytes, represented by the integer values 0 to 255.

The first step is to build the frequency table.  This requires an initial pass through the entire input which keeps track of the number of occurrences of each byte in an integer array. The complexity of this pass in O(n).  We then build a code tree and a canonical Huffman code from the frequency table.  Here we are making a fixed number of passes over fixed length arrays and queues which will be constant time, so the complexity for this part is O(1).  When we write the output, we first write the code table, which is again constant time.  Then we make another pass through the input to do the actual encoding with complexity O(n).  Thus the overall complexity of our Huffman encoding implementation is O(n).

One way in which we can change this algorithm is by eliminating the initial pass through the input to build the frequency table.  This also eliminates the need to build a canonical code tree that is included with the output.  The strategy for this implementation is to start with a blank frequency table and increment the counts for each byte as the input is being encoded.  At first we have no data on frequency so the encoding is suboptimal.  But as we progress through the input and update the frequency table, we periodically rebuild the code tree and use the updated Huffman codes to encode the next series of bytes.  While this means that a particular byte could have a different code at different points in the output, the rebuilding of the code tree is done at the same points in both the encoding and decoding processes so that the output of the encoding is decoded correctly.  The complexity of this version of the encoding algorithm is also O(n), although in practice, since it’s only making a single pass through the input it will take slightly more than half the time of the original implementation.

**Serialization**

As mentioned before, the writing of the compressed file was not actually done to the MP3 standard, but was instead a simplified writing of bytes to a raw data format (.jet). This