Zach Swanson  
04-30-2018  
ECEN-865  
Homework 3 – Huffman Encoding

The purpose of this assignment was to gain practical knowledge of implementing a in-place implementation to generate Huffman codes. The algorithm required three phases:

1. Locate root node
2. Allocate depths to internal nodes
3. Determining codeword lengths of all leaves

Once the codeword lengths were determined, the Huffman codes could be generated by whatever manner was desired. For this assignment, a simple counter method was used, whereby codewords were created in order of increasing length. That is, a codeword of minimum length L was encoded and a counter was increment until the L most-significant bits no longer matched the encoded value.

Many difficulties were had developing the correct indexing for the in-place algorithm. The difficulties continued in determining how to generate the codes from the code lengths and how to handle bit streams. After the core of the algorithm was implemented, errors persisted in so far as the program was not agnostic to file type. The program would compress .txt files with few errors, but when other file types such as .pdf were encoded, the decompressed file was wrought with errors. The culprit was a type-casting error during the encoding stage.

For results, several .txt, .pdf, and .png file were encoded and decompressed. The .txt files had the highest compression ratio: approximately 60% for all files tested. The compressibility of the .pdf and .png files ranged from 92-97%. This makes sense because the .pdf and .png files are mostly compressed already. In terms of errors, the .txt files were typically 98-99% matches to the original file and the .pdf and .png files were 100% matches. The errors in the .txt files were due to nonuniformity in the symbols. Certain symbols arose a handful of times while others arose tens of thousands. Due to the self-imposed restriction of 16-bit encoding, the infrequent symbols required longer codeword lengths and were effectively replaced by dummy codes.