Huffman Encoding and Decoding Progress Report

Group Members:
Salem Ait Ami (V00932871)
salemaitami@uvic.ca

Adam Tidball (V00936605)
adamtidball@uvic.ca

Introduction

The project topic discussed in this report is Huffman encoding and decoding. Huffman encoding and decoding is an encryption technique that utilizes varying length encryption codes, based on each individual symbol's probability of occurrence, to achieve a lower transmission rate. This report provides an overview of the current progress made on implementing Huffman encoding and decoding, as well as plans for future optimizations for this project.

Design Specs

The unique design specifications of our implementation of the Huffman encoding and decoding project are as follows:

- Alphabet Size: We are currently using ASCII symbols as an alphabet and the code can easily be extended to accommodate extended ASCII as well.

- Input Text Size: The program accepts variable file lengths, provided they can be accommodated by an array.

- Hardware: The hardware used for compiling and running the software is the UVIC ARM machines that can be accessed over the intranet. More information on this hardware can be found in the processor section.

Design Blueprint

Our design can be broken into two parts: firstly, the Huffman encoding section, and secondly the Huffman decoding section.

- For the Huffman encoding part of the project a single tree lookup table is created using an alphabet with predefined symbol probabilities. The table's index corresponds to the ASCII value of the character at that index. This feature makes the encoding more efficient, because instead of searching the table for a symbol from the sample text we can simply cast a symbol to an int and access its index. For this reason, the overhead incurred from replicating the data from the min-heap to the table is worthwhile as it allows us to query the table in constant time. Code optimization techniques will then be applied to this design.

- For the Huffman decoding section, three distinct designs will be implemented. Firstly, a naive solution using a brute force approach to the decoding will be used. This should be a relatively easy function to code and will act as a good baseline for comparing decoding efficiency with the second design. The second design that will be implemented uses a variable-length strategy and lookup tables to try and make decoding more efficient. This is the design that will be optimized using code optimization techniques and compared to the results of the naive solution. The third design uses a min-heap implementation of the huffman tree where a cursor traverses the tree by reading the encoding. If the current code is 0, the cursor goes to the left child and if the code is 1, the cursor goes to the right child. If the node is a leaf, then we have successfully decoded a symbol and we print it before setting the cursor to the root for the next iteration.

For the decoding part of the design a minimum heap tree will be created to store the various symbols and their corresponding probabilities. An example of one such distribution of symbols in the tree can be seen below:

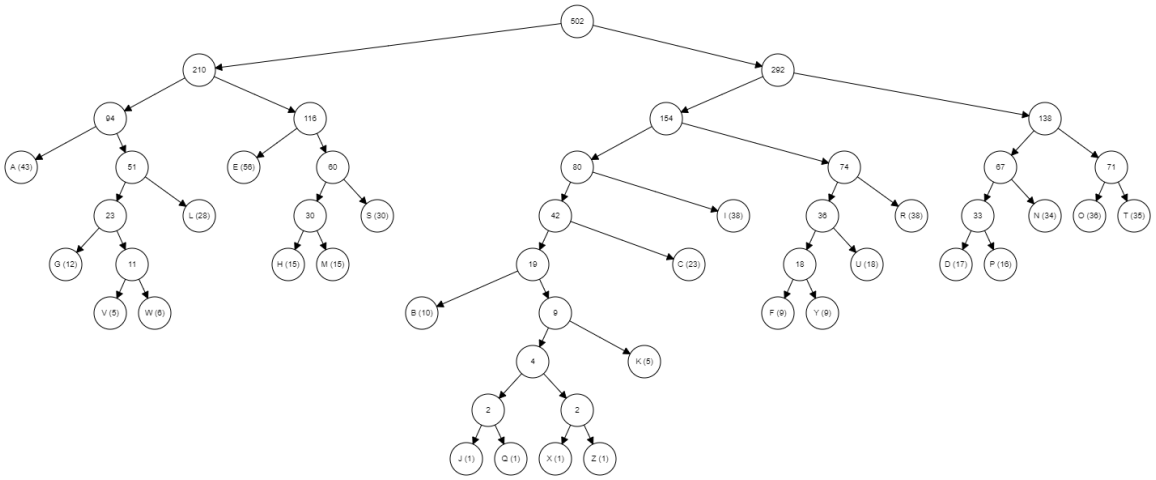

Figure 1 - Example Symbol Probability Tree

For the second part of this project, the Huffman decoding, look up tables will be created based on subsets of the main alphabet probability tree. The optimum number and size of the lookup tables is something that we plan on determining through varying trials and benchmark tests. Although, as a general rule, we know that the table should be small enough so that it can be traversed quickly and remain in cache memory while large enough so that it contains a useful amount of symbol information.

Processor

The processor that we plan to use is the real ARM machine available via the UVIC lab computer within the Engineering intranet. This is a 32-bit ARM processor that has a corresponding word length of 32-bits (4 bytes). We plan to compile the code using arm-linux-gcc both with and without SIMD NEON intrinsics enabled to compare the results.

Work Plan

Currently the basic Huffman encoding and Decoding parts of the project have been implemented, so the next major goal is to get a functional variable-length Huffman decoding solution running. Once this decoding is functional then the focus will shift to optimization, however we suspect variable-length Huffman decoding may be difficult to implement.

Once variable-length Huffman decoding is working the plan is to optimize the decoding process further by running benchmark tests and determining the ideal size and number of tables. As well as performing code optimization techniques in the form utilizing strategies such as loop unrolling, and reducing global variables. This could be a very time consuming process as many benchmark tests can be run as well as many code optimization techniques can be tried.

Finally, research into possible hardware solutions or optimum performance given unlimited resources will be done based on the data from our findings. This is a more calculation intensive and theoretical part of the project that will require a good understanding of the current system shortcomings of which can be improved.

Questions

Some questions we have based on issues that we foresee and have encountered already are:

- For code compilation using the UVIC ARM machines, is compiling the code using arm-linux-gcc both with and without SIMD NEON intrinsics necessary? Or should we focus on just one compiler?
- When creating lookup tables is the approach of using subtrees of the entire alphabet probability tree correct? Or should a different table data structure be used?
- Is it fair to assume the naive decoding solution is the least efficient and use it as a baseline while focusing on optimizing the variable-length decoding solution?
- What is the difference between the naive brute force decoding approach and the barrel-shifter look up table part of the variable-length decoding?
- We have been referencing the following ARM architecture manual: https://www.intel.com/content/www/us/en/content-details/654202/arm-architecture-reference-manual.html?wapkw=ARM%20DDI%200100E. Is this sufficient when using the UVIC machines, or are there other recommended resources?