sievenna Design Document

Onni Aarne

January 21, 2018

1 Structure

sievenna is a command-line application. Parsing of command line options is done by the Apache Commons CLI library. This happens in the Main class. The main method then calls either the encode or decode method of the HuffmanCoder class, which then processes the data appropriately, reading and writing the encoded file using the binary output and input classes.

If the -l flag is enabled, a MinimalistTimer is used to log how long each part of the process takes.

2 File format

The encoded file contains an encoded trie describing the Huffman codes of each possible byte value, an integer indicating the number of bytes in the original file, and finally the Huffman coded file [2].

2.1 Trie encoding

The encoded trie is a full binary trie, meaning all nodes have 0 or 2 children. This enables us to unambiguously define the tree by simply writing the values of all nodes in preorder. This is done by writing a 0 for each non-leaf node and a 1 followed by the key byte for each leaf node. Because there are always 256 leaves and 511 nodes in total, the trie is always written in 2559 bits which is just under 320 bytes.

If the alphabet were to be expanded, the size of the trie would grow linearly with the alphabet.

3 Analysis

3.1 Encoding

The encoding process involves a number of steps:

1. Read file
   This is trivially done in $O(n)$ time and $O(n)$ space.

2. Build model
   The model, which is a probability distribution over the symbols in the file, was built by simply counting the frequency of each byte value in the file, so it took $O(n)$ time and $O(a)$ space.

3. Build trie
   The trie is built by initializing all leaf nodes as their own trees and then repeatedly joining the two tries with the lowest probability under a new root node until there is only one trie with probability 1 [1].

```
build-trie(int[] probabilities)
    H = new minheap
    for i = 0 to 255
        heap-insert(H, new node(key = i, P =
           probabilities[i]))

    while (H.size > 1)
        left-node = heap-del-min(H)
        right-node = heap-del-min(H)
        new-root = new node(
            key = -1,
            P = left-node.P + right-node.P,
            left = left-node,
            right = right-node)
        heap-insert(new-root)
    return heap-min(H) \\ return root of complete trie
```

   The while-loop needs to be run once for each non-leaf node, and it includes the $O(\log a)$ heap-insert, which gives it $O(a \log a)$ time complexity relative to alphabet size and $O(1)$ time complexity relative to input size. The for-loop has the same time complexities for similar reasons, and so does the function overall. The heap takes up $O(a)$ memory, and so does the whole function.

4. Form code table
   The code table is formed by completely traversing the trie, which takes $O(a)$ time.

5. Encode file
   Writing the trie takes $O(a)$ time, and writing the byte count takes $O(1)$ time. When encoding the file body, the codes corresponding each byte can be found in constant time and writing each code takes $O(\log a)$ time because the length of the codes grows logarithmically as the alphabet grows. The total time taken is therefore $O(n \log a)$

The encoding process takes in total $O((n \log a) + (a \log a)) = O((a + n) \log a)$ or simply $O(n)$ when assuming constant alphabet size.

3.2 Decoding

Decoding involves more or less reversing the encoding steps. The trie is read and constructed in $O(a)$ time, after which the file body can be decoded in $O(n)$ time relative to the size of the compressed file, which grows linearly relative to original file size (See testing document). Decoding the file body takes linear time because for each bit read, the trie is traversed one step, and whenever a leaf is reached a byte is written.

4 Future Work

As it is, sievenna only uses a rather naive model of the data and encodes that. Because it does not have a sophisticated model or utilize any sort of transform, it cannot, for example, exploit any repeating patterns in the code [1]. A file containing every byte value equally frequently would not be compressed at all, even if the values followed some obvious pattern such as 0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2,2,3... However, over the course of this project I have learned a fair bit about such methods, even though I did not have time to implement them. I might return to this project to improve it if I have time in the future, even though I don't know that the world needs another command line compression tool.

References

[1] Matt Mahoney. Data compression explained. **mattmahoney. net, updated May**, 7, 2012.

[2] Robert Sedgewick and KD Wayne. Algorithms. 4th, 2011.