CS5002 Project Write-Up

Yao Xiao

December 6, 2016

1 Overview

The program implements Huffman coding, a data compression algorithm that uses fixed-to-variable length encoding to exploit the non-uniform distribution of characters in files.

2 Changes

I implemented everything in the proposal, except the 'if time allows' part, because time was cruel and didn't allow.
In terms of implementation, I made a few changes - I didn't sort the frequency array, I used the data structure linked list which I didn't think of, and I modified the struct I used to represent the leaves. I also implemented the code in Dr. Racket with the help of SICP (Structure and Interpretation of Computer Programming).

3 User Interaction

On the command line, do the following:

```
> make
> ./huffman
```

Follow the system prompt and proceed.

4 Results

I tried out the huffman tree with four files - test_small (421 bytes), test_even (712 bytes), test_mid (1492 bytes), and test_large (3408 bytes).
The compressed test_small file is of size 454 bytes, which is larger than the original file. This is because we also included the tree in the file. test_even stroke a balance - the compressed file is of size 711 bytes. This is not a robust analysis but I suspect that the break-even point is around 700 bytes. Then, when I go over 1KB, the advantage of Huffman coded files become more apparent. File test_mid achieved 21% decrease in size. File test_large achieved 30% decrease in size. The biggest file I tried, test_super (around 11 KB), achieved 36% decrease in size. So this is definitely working!

5 Lessons Learnt

META

1. **In Theory vs. In Code**
   Huge difference! It took me about ten times the effort to implement than to understand the algorithm. It was a humbling experience. And it's a little scary - I knew how the algorithm should work in every step and only had to worry about using C to make it real, and I knew for certain that there is an answer, this is a very basic compression algorithm, and still it took me so long to succeed. I can only imagine what the process is like for real research - it's doing this over and over again, for problems that have no clear solution.

2. **Code Validation**
   Just because the code compiles does not mean it does what you want it to do. Always write tests with synthetic data.

3. **Exploit the strengths of a language**
   Exploit the data structures in different languages - use Racket for easy implementation to solidify understanding, use C for speed and low-level manipulation. Exploit the deterministic nature of Huffman tree when saving the tree.

4. **Simplicity vs. Speed**
   Simple code sometimes has a high price tag (see point 3 for detail).

5. There is nothing more motivating than an impending deadline.

CODING

1. The real challenge of the project was to use the bare-bone data structures provided in C. The ubiquitous list structure in Racket is actually a linked list in disguise. I realized this when I was trying to build an array of pointers in C. I spent a lot of time writing (and debugging) basic functions such as finding the minimum element and removing a element from a linked list. I missed writing one-liners in Racket.

   However, the one-liners in Racket turned out to be run-time landmines - for example, the member? function that checks if an element is in a list is actually O(n) because it traverses the entire linked list. To avoid this in C, I took advantage of the speed of array indexing - I added an array of length 256 in the tree struct to track what elements is in a tree.

   If I were to choose between C and Racket, I would choose C, because even though there are less readily available data structures made for me, I have a lot more control over the run time of the program.

2. This is the first time I worked with a real file in C. That was the most time-consuming part of the project. I found out about ASCII encoding vs. UNICODE. I discovered the usefulness of the seemingly useless bell character. I learnt hexdump. I learnt bit writing. I have come to appreciate simple, bare-bone text editors.

3. I have finally gotten a taste of the evils of scanf() - it leaves stuff in the buffer! So later if you want to get input again, you will be stuck with the junk left by scanf() unless you clean it. I could have done a better job by using fget() but I was lazy so I used a quick fix (thanks Stack Overflow!).

4. I got better at debugging. I got better at accepting the existence of bugs. I especially got better at locating segmentation faults. One thing I did not do during the project was aggressively checking for errors because I was lazy and only did it at the end as a touch-up. It could have saved me hours of debugging.