File Compression

GitHub Repository:

Question: Which type of file can my Huffman tree algorithm compress the most effectively and/or quickest?

Hypothesis: The sparse file is the quickest, and the most compressible. The uniform files will be the least compressible.

Abstract:

In my experiment I will be testing a compression algorithm I wrote. This uses a Huffman tree based method. This method uses the frequency of each character to optimize compression. The way most compression algorithms work is to shorten the string of bits used to represent a character. Other binary tree algorithms use fixed length codes. This means that all the codes are the same length for each character, no matter the frequency. The special thing about a Huffman tree technique is that it has variable length codes, and uses frequency to determine which characters should have the shorter codes than others. Therefore, it should have a better compression rate.

Another Interesting feature of this algorithm is that it has a maximum compression rate of 85.7%. The reason maximum compression is 87.5% because the most a Huffman tree can compress an 8 bit character down to is 1 bit. Do the math and you end up with 87.5%. Maximum compression can only be achieved if the file has only one character.

Based on the data in my experiment, my hypothesis is partially correct. The sparse file was compressed more than the other files, at 86.92%. My hypothesis was also correct that the uniformly distributed files were the least compressed. These files are compressed at a low rate, of 28.25%, and might become negative.

Materials:

- A Computer (My computer is an Apple MacBook with Mac OS X 10.6.8 - see below for more specs)
- The Programs I wrote (Available in my GitHub Repository)

Computer Specifications:

- Processor: Intel Core 2 Duo
- Memory: 2GB 800MHz DDR2 SDRAM
- OS: Mac OS X 10.6.8 Snow Leopard
- Apple MacBook

Procedure

1. Gather materials
2. Generate the files to test
3. Close all other programs
4. Run the compression algorithm on all of your files
5. Record results

Variables:

Constants:

- Computer run on
- Number of times in a row run

Responding Variable:

- Speed/Rate of Compression

Manipulated Variable:

- File Run On

Background Information:

In my experiment I will be testing a compression algorithm I wrote. This uses a Huffman tree based method. This method uses the frequency of each character to optimize compression. The way most compression algorithms work is to shorten the string of bits used to represent a character. Other binary tree algorithms use fixed length codes. This means that all the codes are the same length for each character, no matter the frequency. The special thing about a Huffman tree technique is that it has variable length codes, and uses frequency to determine which characters should have the shorter codes than others. Therefore, it should have a better compression rate.

Background Research Citations:

1. "Data Compression." *Wikipedia*. Wikimedia Foundation, <http://en.wikipedia.org/wiki/Data_compression>.
2. "Huffman Coding." *Wikipedia*. Wikimedia Foundation, <http://en.wikipedia.org/wiki/Huffman_coding>.

Conclusion:

Based on the data in my experiment, my hypothesis is partially correct. The sparse file was compressed more than the other files, at 86.92%.

My hypothesis was also correct that the uniformly distributed files were the least compressed. The reason the uniform files are compressed at a low rate is because the Huffman tree has to give long codes to frequent characters. Therefore, the compression on the file isn't very high, and could become negative if the file has enough size and diversity. Negative compression makes the file bigger rather than smaller.

Based on the data, the algorithm is able to obtain a slightly higher throughput (bits per second) for the larger files. This is due to the fact that there is a certain time for "setup" at the beginning of the program, that is about the same for all the programs. The large file has more bits to spread that time across and therefore will be able to process more bits per millisecond.

The Huffman tree algorithm has a maximum compression rate of 87.5%. The maximum compression is 87.5% because the most a Huffman tree can compress an 8 bit character down to is 1 bit. One divided by eight is .125, converted to a percent and subtracted from 100 you are left with 87.5. The only way to achieve this is to have a file that has all the same characters in it.

The sparse file had close to maximum compression, and for a good reason. The sparse file should be about 99.33% zeros and 0.66% other characters. The percent compression was 86.92%, which is 99.34% of maximum compression.

In this experiment I only tested the performance of the Huffman tree, and limited file types. I wonder how the Huffman tree algorithm would hold up against another compression algorithm? Test it with more file types? Further work could be done testing a wider variety of files. Overall I think my project went well and produced good results about the Huffman tree algorithm.

Example of the Huffman tree algorithm:

Take this string of text “aaaabbbcc”. The first thing the computer does is counts the frequency of each character. This string has 4 a’s, 3 b’s, and 2 c’s. The computer then constructs a table with this data.

Character	Frequency	Code	ASCII Encoding
a	4	0	01100001
b	3	11	01100010
c	2	10	01100011

This is the string without compression, in ASCII character encoding.

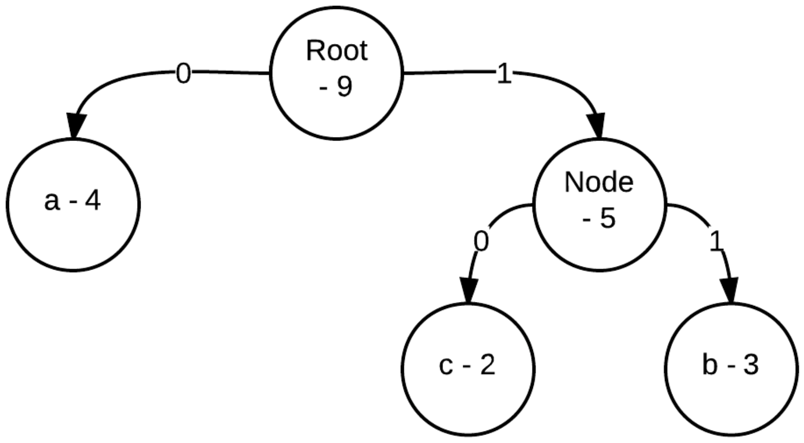
011000010110000101100001011000010110001001100010011000100110001101100011

This is the string with my huffman tree algorithm.

00001111111010

You can see the difference just by looking at the width of the line.

The way the Huffman tree algorithm works is by constructing a tree, as the name implies. To do this the algorithm uses the table of frequencies created earlier, except reversed, with most frequent at the bottom. It takes the first 2, which are the least



frequent, removes them from the list and makes them the child nodes of a new node. It then adds this subtree to the list, in sorted order, using the combined frequencies of the child nodes as it’s frequency. Then it repeats this process until there is only one node left in the list. This is the root node of the tree.

So for our example it would start by making 2 new nodes. One with the character c, and the frequency 2. Another with the character b, and the frequency 3. Then it would make a new node with the frequency of 5 and insert it into the list.

A	4
Node	5

Then the algorithm does it again. Takes the remaining two items in the list and makes them children of a new node, whose frequency is 9. This is the root of the tree.

Now it has a completed tree and it can generate codes from it. To do this it labels each right arm of a node with a 1, and each left arm with a 0. Then it finds the code by following every path to the characters. The codes are in the table above.