Data Structures

Author: Albano Drazhi

Huffman Trees in Compression and Decompression

Project Report

December 12, 2013

Huffman Trees in Compression and Decompression

Abstract

Compression is a technique used to facilitate storing large data sets. The purpose of this paper is to give an introduction about Text Compression and Decompression techniques using a Huffman Tree as a core data structure. The paper also explains how basic compression and decompression is performed, and how to encode/decode written messages. We will also discuss the runtime analysis and advantages/disadvantages of Huffman Compression technique. This paper also gives an idea of the efficient Huffman Implementations that are used in solving critical problems by tech giants such as Google.

I. Introduction

Data compression is the process of reducing the amount of data needed for storage or transmission of a given piece of information (text, graphics, video, etc.), typically by use of encoding techniques. Compression is useful since it helps reduce resources usage, such as storage space of transmission capacity. Compression is used everywhere from the images we get on the Web and modems, to BigTable, MapReduce and RPC Systems [1].

Because we cannot hope to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are most likely than others. In other words, there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of messages - i.e., repeated characters are more likely than random characters. Therefore, Compression is all about probability [2].

Such an algorithm, that assigns short codes to input blocks with high probabilities and long codes to inputs with low probabilities is the Huffman Coding Algorithm. We use such algorithm in this project and Huffman Tree as a core data structure that will be used to obtain the codes from the generated tree.

II. Methods

The Huffman Coding process is applicable to many forms of data transmission. In our implementation we will use text files as a means to perform the coding/encoding process. We will begin by giving a description of our Algorithm. We may interchangeably use the terms encoding and decoding for compression and decompression respectively.

- The basic idea is that the encoding process will have as its main concern scanning the text that will be compressed and keep track of the occurrence of all characters that are part of the given input block.
- Characters will then be sorted or prioritized based on the number of occurrences in the input.
- A Huffman Tree will be built based on the prioritized list of characters
- We then perform a tree traversal and create a coding file that matches with the input.

The nodes will be inserted in a priority queue.

```
public class HTree {
   private Queue<HNode> q;
   ...
}
```

This is a simple process as the lower the occurrence, the higher the priority in the queue.

The encoding process in our project uses an ASCII table to encode each character. The backbone of such a process is the `InStream` class under the project directory which provides the ability to read individual bits to a file. In our approach the number of bits will be a multiple of 8, however this is a minor issue.

After the blocks of input are read, `MainHCGenerator` class generates a mapping

of each character to an `ASCII` representation. The main concern of the mentioned class is streaming through the input file, counting the frequency of each character. The input file results will then be used to build a Huffman Tree and print the codes to a coded file which will be written in the same directory as that of the input file.

The decoding process, as it can be intuitively understood will decompress the compressed file. However, one question that might arise is that how does the `HDecompressor` constructor know what are the codes that match to the given input file. This is explained by what is mentioned in the previous paragraph – that besides the given blocks of compressed data, the Decompressor will also use the generated code file. Both the encoder/-compressor and decoder/decompressor will not attempt to encode and decode a symbol that is not in the given tree, respectively. It is also worth noting that the decoder will only decode text using the same coded file that was used during the compression process. The table below shows a simple demonstration of such process.

Table 1: *Encoding and Decoding cycle*

	Huffman	
	Encoding	Decoding
	Input **Coded**	Input **Coded**
Result	Encoded	Decoded

III. Data Structures

Data structures allow a programmer to structure large amounts of data into conceptually manageable relationships. They also provide guarantees about algorithmic complexity – choosing an appropriate data structure for a task is crucial to the well functioning of the Software Development Lifecycle(SDL). We have chosen to use a Huffman Tree as a core data structure in this project, in order to represent the overall tree of character frequencies.

We will also use a Priority Queue in order to provide fast access to the minimum (or maximum) element. All the nodes will be inserted on the Queue and for as long as the priority queue contains two or more nodes we go through a series of steps some of which are mentioned below:

- Create a new node
- Dequeue node and make it left subtree
- Dequeue node and make it right subtree
- The frequency of the new node will be equal to the sum of frequency of left and right children
- Enqueue new node back to the queue

The Dequeue and Enqueue operations refer to performing a remove and insert operation, respectively.

IV. Development Strategy

We first started on building the Huffman Tree from the map of character counts. Then we worked on creating the map of `char` to `String` encodings from our Tree. Next, we worked on using the encodings to compress files, and last worked on decompressing a file that was previously compressed. For the nodes to be able to be stored in the priority queue, the queue needs to know how to sort them. Therefore the `HNode` class implements the `Comparable` interface. As previously stated, nodes should be compared by character frequency, where a character that occurs fewer times is "less than" one that occurs more often. In this context, if two nodes have the same number of occurrences, they are considered "equal". We also took into consideration writing a `toString` method so we could easily print nodes or priority queues of them. Moreover, the `toString` method in Java, unlike the `println`, it shows the elements in the order they would be returned.

Telling whether a file has been successfully compressed is difficult. A Huffman-compressed file will have a gibberish appear-

ance, since the text editor will try to interpret the bytes as `ASCII`. The `InStream` and `OutStream` classes use most of the core features of Java's `InputStream` and `OutputStream` respectively.

V. Running and testing

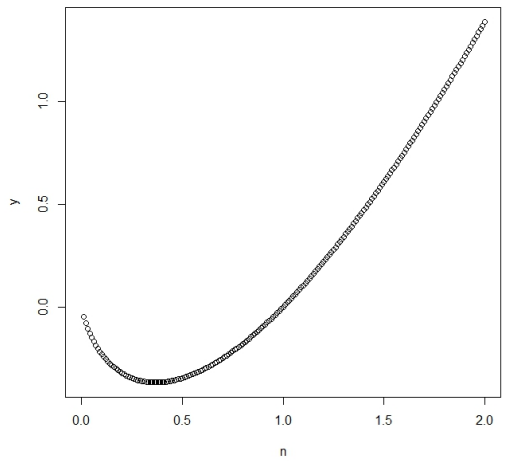
The main class on the package is `MainHCGenerator`. By running it the user will have to interact with the Java console to provide information about name of the input file. Note that the input file should be in the same directory as the project files. The program execution assumes that the user has read/write access to the current directory on the server or local machine.

Also, an important aspect when dealing with data compression is that it's a good practice to test the program with input that comes from Shakespeare's writings or other similar type of material so that the whole process can be more correct.

If time would allow, it would be a good practice to create a GUI for the program as well so that it's optimized for usability. However given the current deadlines, we had functionality and data structure as a priority.

VI. Runtime Analysis

In Huffman Compression, the variable-length codes assigned to input characters are Prefix Codes, i.e., the codes are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is basically how Huffman Compression makes sure there is no ambiguity when decoding the generated bit stream. This implies that Huffman Coding can be constructed using a Greedy Algorithm approach. Sort the symbols in increasing order by frequency. We merge the two least-frequently used symbols `a` and `b` into a new symbol `ab`, whose frequency is the sum of the frequencies of the symbols of its child. By doing the replacing, a smaller set of symbols is left. This process is repeated `n-1` times until all symbols have been merged. The merging operations define a rooted tree, having the leaves containing the original alphabet symbols. Our implementation using a Priority Queue to efficiently maintain all the symbols by their frequency during construction yeilds our Algorithm in `O(nlogn)` time. Below is shown a graph generated in the R Programming language of the runtime complexity of our Algorithm implementation.



VII. Research on Efficient Implementations

As part of the learning process, a good strategy would be doing some research on Efficient Huffman Coding Implementations and observing what are some of the techniques that have helped build the top quality libraries. The work on efficient implementations of Huffman Compressors/Decompressors seemed to take two directions: memory-efficient implementations, or speed-efficient implementations. Memory-efficient implementations tend to save as much memory as possible in order to have the program run with very little memory. Speed-efficient implementations are mostly concerned with the speed of data compression and decompression. Such task may also rely on alternative data structures, however, there is not necessarily a concern about the memory spent. A a good example of the latter would be *Snappy*, a compression/decompression li-

brary. It does not aim for maximum compression, memory-efficiency or compatibility/interactivity with any other compression library, but rather aims for extremely high speeds and reasonable compression. *Snappy* compresses at about `250 MB/Sec` and decompresses at about `500 MB/Sec`. It is widely used inside Google, in everything from BigTable and MapReduce to their internal RPC systems. It is also used in software such as Apache Hadoop.

VIII. Limitations and future improvements

The implemented Huffman Compression and Decompression Algorithms only work with alphabets up to $2^{21} - 1$ symbols, which corresponds to `Integer.MAX_VALUE`. We built the code with simplicity in mind. Thus it is not optimized for performance whether it is memory-efficient or speed-efficient. However, a dynamic programming approach would make the implementation more memory-efficient by dividing the main problem into subproblems, solving each of the subproblems and composing the solution out of small pieces. For speed improvement, a Finite-State Automata could be used, the basic idea being rather than decoding bit by bit which is how thing work in our implementation, decoding of the input will proceed by chunks of `m` bits, and the internal state of the decoder will be represented as the state of an augmented automaton(a NFA with additional types information associated with automation during runtime) [3].

IX. Conclusions

Huffman codes are very popular and frequently used in text compression. We note a large number of papers concerned with efficient implementations, in the contexts of data communications, compressed file systems, archival, and search. However, Huffman codes do have some disadvantages. Two passes must be made over the document on encoding, first to build the coding table, and then to actually encode/compress the document. Second, the coding table must be explicitly stored with the document to decode/decompress it, which result in spending more space. Finally, Huffman Codes only exploit nonuniform symbol distribution, instead of recognizing the higher order redundancies.

References

[1] Snappy - A fast compression/decompression library[Online], Available: https://code.google.com/p/snappy/.

[2] Compression and Huffman Coding.[Online].Available:http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec19.pdf

[3] Steve S. Skiena,"The Algorithm Design Manual" [Hardcover].Springer, November 14,1997.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, third edition.MIT Press, July 2009

[5] Huffman Coding.[Online], Available: http://algs4.cs.princeton.edu/55compression/Huffman.java