716181 Algorithm Design & Analysis

Coding Algorithms

Team I

Charlotte Paterson 1243532
Nancy Vainikolo 1241404
Ximei Liu 1421604
Yue Li 1251124

10th November 2014

Contents

1 Abstract

Coding algorithm is an important topic in computer science and software development. Algorithm efficiency in terms of data compression has been improved by many researchers and developers. The interest of this report is to look into both Huffman Code and Arithmetic Code in depth to determine which is more efficient in data compression. This paper will examine the performance between both algorithms to discover which is superior in efficiency.

2 Introduction

Computer technologies have lifted many aspects of computer usage. For example, computers are increasingly using faster processing power than previous generations and allow more Internet users to use Internet concurrently. The transactional efficiency over the Internet becomes a major concern because large data flow can cause traffic congestion. An approach towards this issue is to compress data into a smaller size to reduce data transactions.

If all users transfer files in their original size over Internet, the processing time would be very inefficient and the Internet throughput would be very low. Also storing files of the original size could cause a lot of storage space to be unnecessarily occupied, especially when most files have a lot of redundancy. Therefore, if data could be effectively compressed wherever possible, significant improvement of data throughput and resource reduction can be achieved.

This report will introduce two lossless data compression algorithms: Huffman Coding and Arithmetic Coding. This paper will firstly introduce the concept and entropy and give the absolute lower bound of lossless data compression. Then the following section will demonstrate how Huffman and Adaptive Arithmetic Coding work with specific examples. This paper will also examine the time complexity of both algorithms. Finally it will compare the performance of Huffman and Arithmetic coding from data compression rate, encoding/decoding efficiency and application.

3 Variable Length Code

Variable length Code is generally used to represent symbols. The length of the code for the symbol becomes shorter when the symbol has higher probability.

$$P(A) = 0.5, P(B) = 0.25, P(C) = 0.25$$

r Now we would encode using bits. As "A" has most occurence frequency we replace it with a smaller bit.

$$A = 0, B = 01, C = 10$$

3.1 Prefix Code

Prefix code is a type of code system which states that no valid codeword is a prfix of any other valid codeword (Alvarez). In a binray tree, it means each symbol is a leaf of the tree.

Any encoding that satisfies this property is known as a prefix code. Huffman encoding will yield an optimal prefix code.

4 Entropy: Shannon's Coding Theorem

The theoretical lower limit for the number of bits needed to encode a probabilistic data source was discovered by Claude Shannon in his classic paper "A mathematical Theory of Communication".In the paper, Shannon stated a source coding theorem, and found that the concept of entropy from statistical mechanics plays a key role. Entropy evaluates the degree of disorder in a mechanical system, which is related to the number of possible states of the system. Shannon's lossless coding theorem is used to determine the best possible result without loss of data information by "establishing a tight lowerbound on an average code length" (Alvanez). The theorem states that "any lossless compression technique must use at least as many bits per symbol on average, as the entropy type data source" (Alvanez).

$$H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

If we take the probabilities in the example of the previous section and insert them into Shannon's entropy theorem, the entropy of this data source is

$$H = 0.5log_2 2 + 0.25log_2 4 + 0.25log_2 4 = 1.5bits/symbol$$

From this we can see that it would take at least 1.5 bits to encode each symbol from the source on average.

5 Huffman Code

The idea of Huffman coding is based on the idea that a variable length code use the shortest codewords for the highest frequency symbols and the longest codewords for the lowest frequency symbols (Alvanez).

5.1 The algorithm

Huffman coding is one of the best known algorithms for lossless data compression. It's a technique that aims to reduce the size of data by eliminating redundancies (Ling & Xing, 2004). The algorithm relies on a binary tree structure. Every branch in the tree determines a coded symbol while the nodes in the branch accordingly represent the codeword for each symbol (Tanka, 1987). The nodes are initially leaves and contain information such as the "symbol itself, the weight (frequency of appearance) of the node and optionally a link to a parent node" (Sharma, 2010). Internal nodes also contain links to two children. In our implementation, a priority queue is used to hold the weight values of the nodes to ensure the

lowest weight is always kept at the top of the queue (Sharma, 2010). An ArrayList holds initial weights appointed to the associated leaves. In Mamta Sharma's paper on "Compression Using Huffman Coding" steps on creating the subtrees are seen as follows.

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the priority queue (by frequency in increasing order so that the least likely item is in the head of the queue).
3. While there are more than one node in the priority queue:
   (a) Dequeue the two nodes with the lowest weight.
   (b) Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the weight of the internal node.
   (c) Enqueue the new internal node into the priority queue.
4. The remaining node is the root nodeand the Huffman Tree has now been generated.

Therefore overall Huffman's algorithm starts with the leaves of the tree and merges nodes with least weight repeatedly. Here is pseudocode that shows the algorithms implementation.

$Huffman\ Coding$
$Input:\ Array\ f[1...n]\ of\ numerical\ frequencies.$
$Output:\ Binary\ coding\ tree\ with\ n\ leaves\ that\ has\ minimum\ expected\ code\ length\ for\ f.$
$Huffman(f[1...n])$

1. $L\ =\ insert\ the\ frequencies\ into\ a\ List\ for\ all\ symbols\ in\ the\ message$
2. $Q\ =\ insert\ all\ the\ entries\ from\ the\ list\ to\ a\ Priority\ Queue$
3. **while** $size\ of\ Q\ >\ 1$
4. $i\ =\ removeMin(Q)$
5. $j\ =\ removeMin(Q)$
6. $f[n+k]\ =\ f[i]+f[j]$
7. $insert\ parent\ node\ f[n+k]\ back\ to\ the\ priority\ queue$

Figure 1: Construct Huffman Tree (Alvarez)

5.2 Example

For instance, "BECABEE" becomes an 8 bytes file (symbol 'EOF' is used to indicate the end of the file. in ASCII, each char becomes a byte, consider 0-255 representation of all symbol codes). There are 3 'E's, 2 'B's, 1 'C', 1 'A' and special symbol 'EOF' with ocurrence 1.

As 'E' is the most frequent symbol in the message, it's assigned the least bit as 0. The "B" is assigned with 10. The 'C' is assigned 110, 'A' is assigned 1110 and 'EOF' is assigned 1111.

Combine the information, we get 100110111010001111. This bit stream has length of 18. Due to byte conversion, the file size will be roughly between 2 to 3 bytes. In actual practice, the file will be 3 bytes.

To decode, we need to reconstruct the original Huffman tree. We read the bit stream 100110111010001111. The bit reader reads each bit. If there is a corresponding bit stream that has a symbol representation, the bit stream indicates a leaf node. After then, convert the bit stream to the symbol and remove the used bit stream. If the bit stream does not have a symbol representation, continue the process to the next bit. After the decode process we can get "BECABEE".

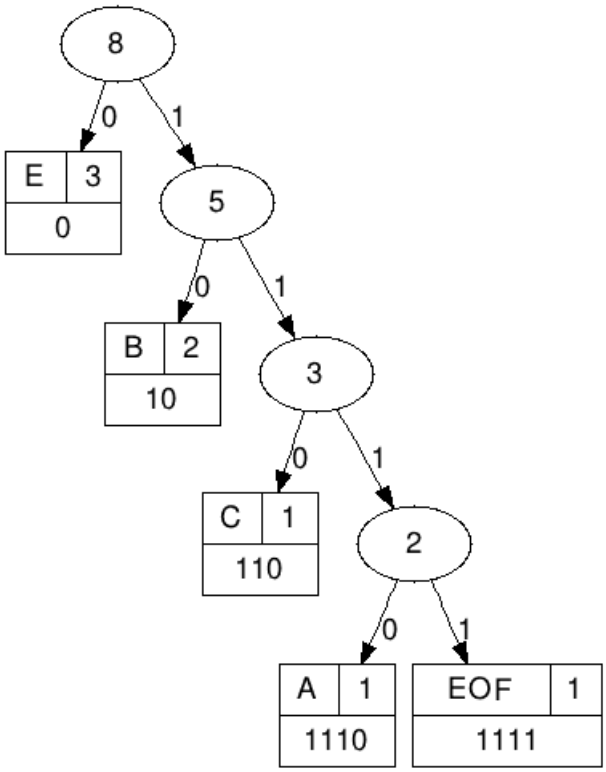

Figure 2: Huffman Tree example

5.3 Time complexity

The time complexity for Huffman algorithm is determined by how the priority queue is implemented. In our implementation for Huffman Code, we chose the PriorityQueue in java.util package. Due to the time complexity of the priority queue operations are $logN$, and to construct the Huffman tree, it needs linear time operations. Therefore, the overall time complexity for Huffman Code is $O(NlogN)$.

5.4 Discussion

The prefix code generated by Huffman coding is optimal in the sense of minimizing the expected length among all binary prefix codes for the input alphabet (Alvarez). However, it does not mean that Huffman code achieves the Shannon Limit.

Considering a binary data source with the following probability:

$$P(A) = \frac{127}{128}, P(B) = \frac{1}{128}$$

The entropy of the data source is:

$$H = -(\frac{127}{128}log_2 \times \frac{127}{128} + \frac{1}{128}log_2 \times \frac{1}{128}) = 0.0659$$

So the entropy of this data source is just about 0.07 bits per symbol; however, Huffman encoding for it will use 1 bit per symbol and hence is clearly far from Shannon Limit.

Huffman coding requires the symbol probabilites or frequencies in advance and each symbol needs to be translated to integral number of bits. Huffman coding performs optimally if all symbol probabilities are integral powers of $\frac{1}{2}$, but in practice it is not the case. The worst case happens when a data source has one symbol with probability approaching 1, so it may take up to one extra bit per symbol(Perl, Maram, & Kadakuntla, 1991).

6 Arithmetic Code

In this section, the principle of arithmetic coding will be presented. Firstly, we introduce the basic arithmetic coding based on the work of Witten et al. (1987). Due to the drawbacks of this algorithm, we finally choose the adaptive arithmetic to achieve our implementation.

6.1 Basic algorithm

Howard & Vitter (1992) described the algorithm as follows:

1. Get the probability of each symbol in the file.
2. Begin with an initial interval [LOW, HIGH), that's [0,1).
3. For each symbol of the file, two steps will be performed:
   (a) Divide the current interval into subintervals, one for each possible symbol. The size for each symbol's subinterval is proportional to the probability of the symbol.

(b) Select the subinterval that assigns to current symbol and use it as the interval for next symbol in the file.

4. Output enough bits to represent any point in the final interval.

The length of the final subinterval is obviously equal to the product of probabilities of each symbols, which is the probability $p = p_1 \times p_2 \times p_3 \times ... \times p_n$ (p_i is the probability of the ith symbol in the file). The final step need to use $-log_2 p$ bits to present the message in the file. We also need an EOF to indicate the end of the file. We use the following formula to encode symbol a_i

$$Range = HIGH - LOW$$
$$CDF(n-1) = \sum_{k=1}^{n-1} p_k$$
$$CDF(n) = \sum_{k=1}^{n} p_k$$
$$LOW = LOW + Range \times CDF(n-1)$$
$$HIGH = LOW + Range \times CDF(n)$$

6.2 Example

Here we reuse our previous example in Huffman section "BECABEE":

$$P(B) = 0.25, P(E) = 0.375, P(C) = 0.125, P(A) = 0.125, P(EOF) = 0.125$$

Initially, Low=0, High=1, current interval is [0, 1). The encoding processes are recorded at **Table 4**.

The final interval without rounding is [0.13005375862121582031 25, 0.1300601959228515625), which in binary is [0.00100000101001011 01101, 0.0010000101001011 101). We can uniquely identify this interval with output 00100001010010110. Since the probability p of this particular file is shown as:

$$(0.25)^2 * (0.375)^3 * (0.125) * 0.125 * 0.125 = 0.000006437301635742 1875$$

The result is the exactly value of the size of this interval. Information contained in the interval is given by $-log_2 p = 17.2451$. This value also describes the number of bits necessary to encode the whole message in the file. In practice we have to output 17 bits. Compare this result with the one from Huffman, Arithmetic Code uses 1 bit less.

There are some major problems with the above coding:

1. Large computational complexity is a barrier to the above basic arithmetic coding because the additions make it slow, multiplications make it very expensive, and divisions make it impractical.

2. The shrinking of current interval requires high-precision floating point operations.

3. No output is produced until the entire file has been read.

6.3 Adaptive Arithmetic integer implementation

Due to the drawbacks of basic arithmetic coding, we adopted adaptive integer arithmetic to implement the coding. In integer implementation, current interval will represent in sufficiently long integers(32-bit long) rather than floating point or exact rational numbers. Symbol probabilities will be presented with frequency counts instead of actual probability. The subdivision process involves selecting non-overlapping intervals with lengths approximately proportional to the counts. We use the following modified formula to encode symbol a_i.

$Range = HIGH - LOW + 1$
$CDF(n) = \sum_{k=1}^{n} f_k/N$ (N is the total number of symbols)
$CDF(n-1) = \sum_{k=1}^{n-1} f_k/N$
$LOW = LOW + Range * CDF(n-1)$
$HIGH = LOW + Range * CDF(n) - 1$

Since we use semi-open sub interval, HIGH should be less than the LOW of the next interval, and the best integer value for HIGH is HIGH= (next LOW)-1. At the same time [HIGH, HIGH+1) still belongs to the current interval, that's the reason why using for +1 in Range and -1 in HIGH.

6.4 Time complexity

In the code implementation, an array is used to store the cumulative frequency of each symbol. In each iteration, the program needs to look up the array to get the value of $CDF(n-1)$ and $CDF(n)$. In addition, the operation just needs two multiplications and divisions. In order to encode, the program needs to run n times, so the overall time complexity is O(n).

7 Comparison

Under our time constraint, we managed to implement our own Huffman Coding. However, the results for Arithmetic Code are based on a project called Arithmetic Code on GitHub. (by Nayuki Minase under MIT license). We modified the code for analytical purposes.
The table below shows data compression rates between Huffman and Arithmetic Code. Sample 1 is prepared in different combinations of letters. In Sample 1, letters sometimes can have high occurence such as "aaaaaaaaaca". Sample 2 is extracted from a book *Linux Command Line and Shell Scripting Bible* by Richard Blum and Christine Bresnahan. The compression ratio is computed as:

$$Compression\ Ratio = \frac{File\ size\ after\ compression}{File\ size\ before\ compression} \tag{1}$$

Table 1: Huffman/Arithmetic Compression Rate

Data	Sample Size	Huffman Code	HC Ratio	AC(byte)	AC Ratio
Sample 1	765,808	411,417	53.7%	223,090	29.1%
Sample 2	1,324,279	802,776	60.6%	796,744	60.1%

Table 1 shows that when the message contains multiple repetition, Arithmetic coding gave a better data compression ratio than Huffman coding. When the data source is just normal text, Arithmetic coding and Huffman coding give a similar compression ratio.

Table 2: Huffman/Arithmetic Compression Speed

Data	Sample Size	HC(encode)	HC(decode)	AC(encode)	AC(decode)
Sample 1	765,808	1001.243	1520.574	475.940	548.127
Sample 2	1,324,279	1804.973	2565.965	797.375	967.116

Table 2 shows that from the two samples, arithmetic coding is more efficient in encoding and decoding than Huffman. The result matches with the theoretical induction that the time complexity of arithmetic coding implementation is O(N) while Huffman coding implementation is O(NlogN).

Table 3: Huffman/Arithmetic applications (Blelloch 1997)

Application	Huffman Code is used	Arithmetic Code is used
WINZIP, GZIP	✓	
BZIP2	✓	
PKZIP	✓	
JPEG	✓	✓
PNG	✓	
MPEG	✓	✓
MP3	✓	
AAC	✓	
JBIG		✓
Flash		✓
PPM		✓
PAQ		✓
DjVu		✓
Skype		✓

From **Table 3**, it can be seen that Huffman Code is widely used in many general purpose of data compression application. The table also shows that Arithmetic Code is used in applications that can create fairly high compression ratio such as PAQ. According to Blelloch (1997), Huffman is popular among general data compression methods. However, with the patent expiry of Arithmetic Code, Arithmetic will be increasingly used in new software products.

8 Summary

In this report, we demonstrated how Huffman and adaptive Arithmetic code works. We discovered that Huffman coding was much easier to implement than the complex structure of adaptive Arithmetic coding. However, the data we collected from our two sample files, show that adaptive Arithmetic Code indeed has better performance than Huffman Code in terms of efficiency, which was not surprising after further research into the time complexities. We also learnt that program implementations also can have great impact on algorithm performance. This was concluded from the data we recieved based on the compression rate.

Ultimately, based on our focus towards the efficiency of these compression algorithms, the adaptive Arithmetic method is considered more efficient than the Huffman approach. If we had the opportunity to do a more in-depth study on the implementation of the original basic Arithmetic coding method, the results could possibly be in Huffman codings favour. Also to provide a more defined solution, a broad selection of file formats could be tested in order to help give a thorough result between the algorithms. However overall we discovered from the data we tested that although the algorithms provide similar compression rate results, the adaptive Arithmetic method proves to be superior in compression efficiency.

Table 4: Arithmetic Code Interval table

Current Interval	Action	Range	Input
[0,1)	Divide	B→[0,0.25) E→[0.25,0.625) C→[0.625,0.75) A→[0.75,0.875) EOF→[0.875, 1)	B
[0, 0.25)	Divide	B→[0,0.0625) E→[0.0625, 0.15625) C→[0.15625, 0.1875) A→[0.1875, 0.21875) EOF→[0.21875, 0.25)	E
[0.0625,0.15625)	Divide	B→[0.0625, 0.0859375) E→[0.0859375, 0.12109375) C→[0.12109375, 0.1328125) A→[0.1328125, 0.14453125) EOF→[0.14453125, 0.15625)	C
[0.12109375, 0.1328125)	Divide	B→[0.12109375, 0.1240234375) E→[0.1240234375, 0.12841796875) C→[0.12841796875, 0.1298828125) A→[0.1298828125, 0.13134765625) EOF→[0.13134765625, 0.1328125)	A
[0.1298828125, 0.13134765625)	Divide	B→[0.1298828125, 0.1302490234375) E→[0.1302490234375, 0.13079833984375) C→[0.13079833984375, 0.1309814453125) A→[0.1309814453125,0.13116455078125) EOF→[0.13116455078125, 0.13134765625)	B
[0.1298828125, 0.1302490234375)	Divide	B→[0.1298828125, 0.129974365234375) E→[0.129974365234375, 0.1301116943359375) C→[0.1301116943359375, 0.130157470703125) A→[0.130157470703125, 0.130157470703125) EOF→[0.130157470703125, 0.0000457763671875)	E
[0.129974365234375, 0.1301116943359375)	Divide	B→[0.129974365234375, 0.130008697509765625) E→[0.130008697509765625, 0.1300601959228515625) C→(0.1300601959228515625, 0.130077362060546875) A→[0.130077362060546875, 0.1300945281982421875) EOF→[0.1300945281982421875, 0.1301116943359375]	E
[0.130008697509765625, 0.1300601959228515625)	Divide	B→[0.130008697509765625, 0.13002157211303710937 5) E→[0.13002157211303710937 5, 0.13004088401794433593 75) C→[0.13004088401794433593 75, 0.13004732131958007812 5) A→[0.13004732131958007812 5, 0.13005375862121582031 25) EOF→[0.13005375862121582031 25, 0.1300601959228515625)	EOF

References

Alvarez, A.S. (n.d.). Algorithms: Notes on lossless data compression and Huffman coding. http://cs.bc.edu

Blelloch, G. (1997) Algorithms in the real world. http://www.cc.gatech.edu

Howard, P. G., & Vitter, J. S. (1992). *Practical implementations of arithmetic coding*. Rhode Island: Springer.

Ling, S., & Xing, C. (2004). *Coding Theory: A First Course*. New York: Cambridge University Press.

Lombo, C. (2003). Predictive data compression Using Adaptive Arithmetic Coding. *Louisiana State University, Department of Electrical Engineering.* http://etd.lsu.edu

Natarajan, S. (2001). *Entropy, coding and data compression* (6th ed.). doi:10.1007/BF02837736

Perl, Y., Maram, V., & Kadakuntla, N. (1991). *The cascading of the lzw compression algorithm with arithmetic coding*

Sharma, M. (2010). Compression using Huffman coding. *IJCSNS International Journal of Computer Science and Network Security.*

Tanka, H. (1987). Data Structure of Huffman Codes and Its Application to Efficient Encoding and Decoding. *IEEE Transactions on Information Theory, 33(1).* http://paper.ijcsns.org

Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM, 30*(6), 520-540.