DATA COMPRESSION

Enrollment Numbers:- 071407, 071447, 017438

Name:- Rajesh Upadhayay, Ankit Rai, Nikhil Rana

Name of Supervisor:- Mr. Ravindara Bhatt

May – 2011

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
WAKNAGHAT

Table of Contents

<u>CERTIFICATE</u>

This is to certify that the work titled **DATA COMPRESSION** submitted by **Rajesh Upadhayay, Ankit Rai, Nikhil Rana** in partial fulfillment for the award of degree of Bachelor of Technology (I.T.) of **JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT** has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature in full of Supervisor: ___________________________________

Name in Capital block letters**:**______**Mr. Ravindara Bhatt**__________

Designation: **<u>Senior Lecturer Department of Computer Science and IT, JUIT</u>**

Date: **<u>19th May, 2011</u>**

ACKNOWLEDGEMENT

Many people have contributed to this project in a variety of ways over the past few months. We also acknowledge the many helpful comments received from other teachers of the departments and visualization courses and seminars. We are indebted to all those who provided reviews and suggestions for improving the materials and topics covered in our package .We would like to thank our supervisor **Mr. Ravindra Bhatt** for the invaluable advice and positive encouragement he has provided us throughout the course of this project and we are also indebted to **Prof. S.P.Ghrera** , HOD, CS/IT for the constant support and help. We extend our apologies to anyone we may have failed to mention.

Rajesh Upadhyay

Nikhil Rana

Ankit Rai

SUMMARY

Software of compression and decompression of the text files is a window application with which one can store the data in the memory of system in code format which will consume less space or memory then original file. This software uses algorithm to generate code of variable in the string, the main idea behind this is to remove data redundancy to save space.

Why Do People need to compress data?

The main purpose of compression of data is to store the same data in less memory which help in saving the memory of system. Compressed data can also be transferred through internet easily because of its small size.

The software we built has been made using C language by using the logic of Huffman coding method.

The software we built can be use by any type user with just a basic knowledge of operating system. This software can be used even by student to save their system memory. Every user would like to store their data in less space, so that they can store more data in the same amount of memory.

LIST OF FIGURES

1 Introduction

As the world evolves to rely more and more on computers and data storage, data compression becomes an increasingly useful and important tool. Most data used by humans contains a high amount of redundancy. Compression algorithms aim to eliminate this redundancy, while still preserving all the information contained in the original data. The end result is a reduction in storage requirements, in exchange for increased computation.

One of the most well-known and fundamental compression algorithms is the technique known as Huffman Coding, invented in 1952 by David Huffman. It forms the basis for innumerous theoretical and real-world compression schemes, and serves to prove several important properties of compression in general. Unfortunately, Huffman Coding has shortcomings. It always requires an integral number of bits to represent a given symbol; as a result, inefficiencies arise when the optimal number of bits would be fractional. A technique known as Arithmetic Coding has gained prominence as a replacement for Huffman Coding. It avoids the pitfalls of integral bit counts by representing the compressed data conceptually as a single floating-point number.

This paper aims to present and analyze both algorithms.

1.1 Overview of compression's purpose and history

Compression is used in a wide-ranging variety of applications, from the transmission of science data collected on board NASA space probes, to the storage of digital music on personal computers. Almost as long as there has been digital data, there has been compression of that data.

The basic aim of compression is to sacrifice time and/or processing power in exchange for a reduction in storage requirements. This paper concerns itself in particular with lossless compression, which requires that the compressed data (when uncompressed) restores to an exact copy of the original.

1.2 Summary of significant principles in information theory

It is worth noting that there are very strict limits to lossless compression. In particular, no compressor is capable of compressing all data. If it compresses some input streams into a smaller output stream version, it is mathematically provable that there *must* exist other input streams which actually get larger when fed through the compressor!

Thus lossless compression schemes attempt to compress “typical” data streams, which are a particular subset of all possible input streams. Just what constitutes “typical” depends on your application… thus the abundance of different compression methods.

2 Huffman Coding

2.1 Overview

Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

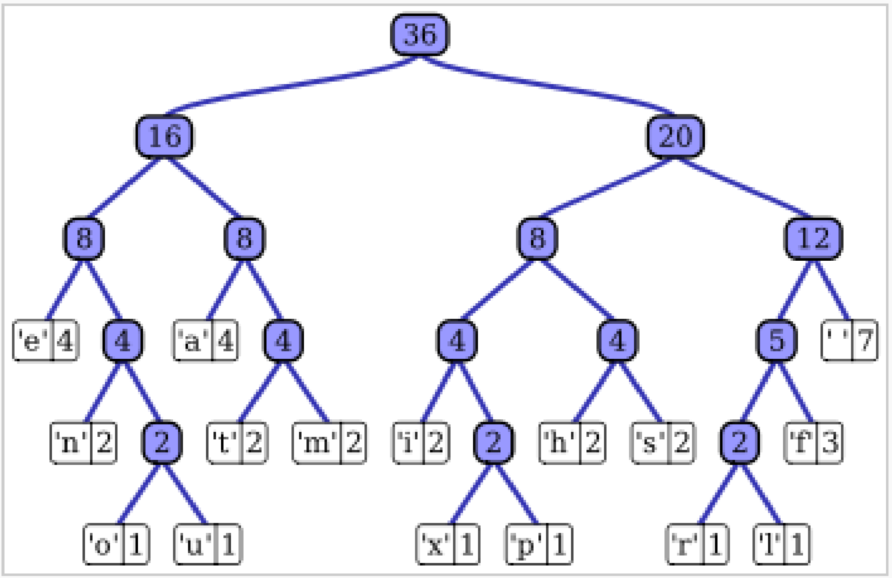
Huffman Coding is a variable-length character encoding which uses the frequency of symbols in an input string to determine the optimal number of bits for representing each character in the output. It requires an initial scan of the entire input stream to determine the input symbol frequency distribution. This stage builds a tree to represent the optimal unique prefix code of each character. The actual compression is then performed by simply applying the translation given by the prefix code tree.

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code(sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol) that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method *of this type*: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code. A method was later found to design a Huffman code in linear time if input probabilities (also known as *weights*) are sorted

For a set of symbols with a uniform probability distribution and a number of members which is a power of two, Huffman coding is equivalent to simple binary block encoding, e.g., ASCII coding. Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e. a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown, not identically distributed, or not independent (e.g., "cat" is more common than "cta"). Other methods such as arithmetic coding and LZW coding often have better compression capability: both of these methods can combine an arbitrary number of symbols for more efficient coding, and generally adapt to the actual input statistics, the latter of which is useful when input probabilities are not precisely known or vary significantly within the stream. However, the limitations of Huffman coding should not be overstated; it can be used adaptively, accommodating unknown, changing, or context-dependent probabilities. In the case of knownindependent and identically-distributed random variables, combining symbols together reduces inefficiency in a way that approaches optimality as the number of symbols combined increases.

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010



Huffman tree generated from the exact frequencies of the text "this is an example of a huffman tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 bits. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

2.2 Precise algorithm description

There are two stages to the static Huffman Coding algorithm. The first is an analysis state to determine the coding scheme, the second is the actual encoding stage.

The first step to determining the coding scheme is to make a single pass over the entire input stream, recording the frequency of each input symbol encountered. Each symbol is then formed into a trivial tree with one node. The number of times a particular symbol occurs in the input stream will be referred to as its tree's weight. One must then perform the following iterative algorithm:

1. T := collection of initial one-node trees.
2. Remove from T elements x, y: the two smallest-weighted trees.
3. Create a new tree z, whose root node has x and y as children.
4. Assign z the weight of x and y combined.
5. Insert z into T.
6. Repeat from step 2 until T contains only a single tree. Call this tree t.
7. For every internal node of t, arbitrarily assign a 1 to one of its child branches and a 0 to the other child branch.

At the end of the above algorithm, t is a full binary tree representing the optimal prefix coding for our input data. Its leaf nodes are the input symbols. The encoding for each symbol is produced by traversing the tree from the root node to the symbol, noting the sequence of 0 and 1 branches along the way.

Thus the compression stage simply iterates through the input stream symbol-by-symbol, outputting the prefix code from the tree t for each input symbol encountered.

The output format from the compressor must also have some mechanism for storing the symbol table, so that the decompressor can perform the compressor's final stage in reverse. Since the symbols occur only on the tree's external nodes, their encodings have the unique prefix property: none their codes is the result of another symbol's code appended with extra bits. The unique prefix property ensures that the decompressor can always decode successive symbols without ambiguity by simply recognizing each compressed symbol one by one, even though they are encoded with varying lengths. The decompressor thus operates by making a single run over the compressed data, performing a lookup on each symbol to translate back to its uncompressed form.

EXAMPLE: The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

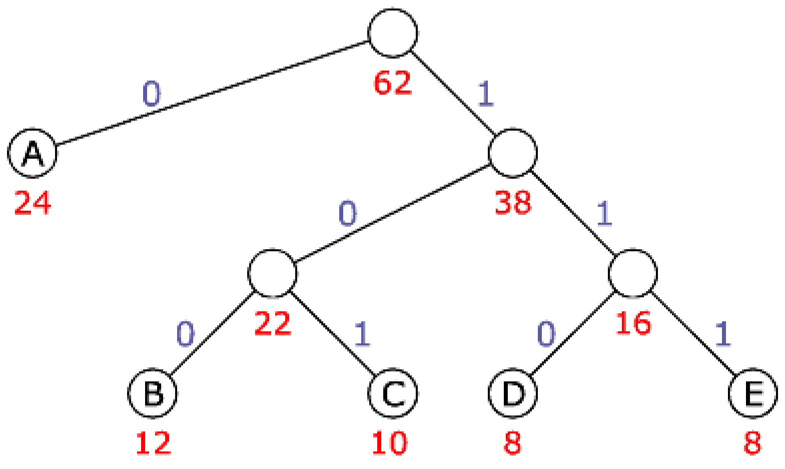
Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

----> total 186 bit

(with 3 bit per code word)

The two rarest symbols 'E' and 'D' are connected first, followed by 'C' and 'D'. The new parent nodes have the frequency 16 and 22 respectively and are brought together in the next step. The resulting node and the remaining symbol 'A' are subordinated to the root node that is created in a final step.

2.3 Code Tree according to Huffman



Symbol	Frequency	Code	Code Length	total
		Length		
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

----

ges. 186 bit   tot. 138 bit

(3 bit code)

Characteristics of Huffman Coad-Huffman codes are prefix-free binary code trees, therefore all substantial considerations apply accordingly .Codes generated by the Huffman algorithm achieve the ideal code length up to the bit boundary. The maximum deviation is less than 1 bit.

2.4 Example:

Symbol	P(x)	I(x)	Code Length	H(x)
A	0,387	1,369	1	0,530
B	0,194	2,369	3	0,459
C	0,161	2,632	3	0,425
D	0,129	2,954	3	0,381
E	0,129	2,954	3	0,381

----

theoretical minimum: 2,176 bit

code length Huffman: 2,226 bit

The computation of the entropy results in an average code length of 2.176 bit per symbol on the assumption of the distribution mentioned. In contrast to this a Huffman code attains an average of 2.226 bits per symbol. Thus Huffman coding approaches the optimum on 97.74%.

An even better result can be achieved only with the arithmetic coding, but its usage is restricted by patents.

2.5 Caveats and implementation details

Generally speaking, the collection T is best implemented as a priority queue. After step 7 has completed, the tree should generally be converted to a direct look-up array. Beyond these simple observations, implementation of the basic Huffman coding algorithm is fairly trivial and follows directly from the supplied pseudo-code.

2.6 Real-world optimizations and extensions

We have a large number of degrees of freedom induced by step 7. One of the most common optimizations is to more carefully choose which children are assigned a 0 and1.

which are assigned a 1. This allows for compact encoding methods in outputting the lookup table, thereby reducing the total output size.

The above algorithm was invented and finalized many decades ago; however active research is still being pursued in the area of dynamic encoding. The algorithm described is actually known as static Huffman coding. More flexible schemes can be used which involve modifying the symbol frequencies during the encoding phase, to adapt to a changing data stream. It should be obvious to anyone that having a single set of symbol frequencies for an entire input data stream isn’t necessarily the optimal solution… dynamic encodings serve to explicitly address this issue.

2.7 Best and worst case performance

Suppose that n is the size of our input stream, and that m is the number of symbols in the alphabet for that stream. Then the initial frequency analysis scan clearly requires $\Theta(n)$ operations. The dominant operations in the loop over steps 2-6 come from the priority queue insertions and removals. Efficient implementations of a priority queue allow these operations to be performed in $O(\log m)$ worst case and $\Omega(1)$ best case operations. A conversion of the tree to an array (after step 7) can be performed in $\Theta(m \log m)$ operations. Thus the analysis stage of the algorithm requires $O(n+m \log m)$ worst case and Ω(n) best case operations. In practice we worry mostly about compression of data for which the size of the input is orders of magnitude larger than the size of the alphabet. For all practical purposes, the analysis stage therefore runs in Θ(n) time.

If a look-up array is used after step 7, then the actual encoding stage trivially requires Θ(n) time, yielding a final Θ(n) time for the compression algorithm as a whole.

The decompression algorithm trivially requires Θ(n) time as well.

As far as compression ratios go, the best compression is achieved with an input consisting of all the same symbol, repeated n times. In this case the symbol is encoded with a single bit, and so the output encoding requires n bits. The worst compression occurs when the symbol frequencies form a Fibonacci sequence. In such a situation, the prefix code tree ends up completely lopsided, and so the ith most-likely symbol requires i bits for its output representation. Using F_i to represent the ith Fibonacci number, the ith most-likely symbol occurs F_i times in the input, and therefore the total worst-case size of the output stream must be:

$$\sum_{i=1}^{n} iF_i = \frac{\phi - (n+1)\phi^{n+1} + n\phi^{n+2}}{(\phi - 1)^2 \sqrt{5}}$$

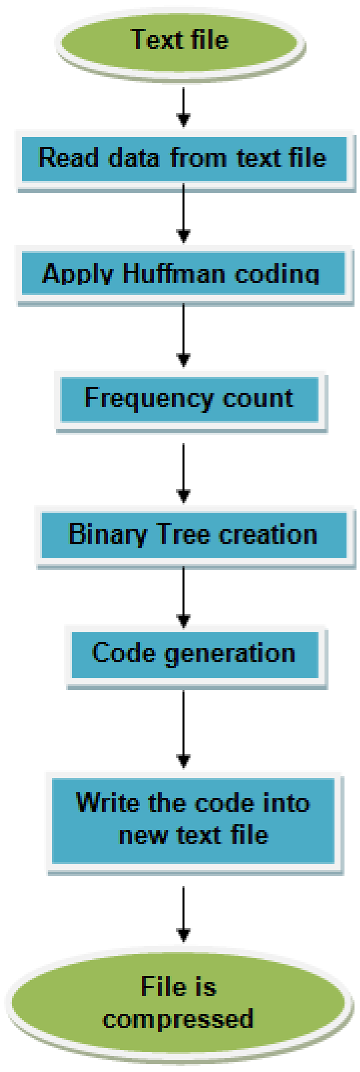
This expansion follows from the fact that $F_i = \left[\frac{\phi^i}{\sqrt{5}} \right]$, where $\phi = \frac{1}{2}(1+\sqrt{5})$, the golden ratio. The dominant term in the expanded summation is easily seen to be $O(\phi^n)$, which gives us some feel for the worst-case compression performance.

2.8 Theoretical significance

The Huffman scheme for generating prefix codes is in fact the optimal prefix encoding scheme. Prefix codes had been studied for many years, but Huffman coding was the first scheme which was provably optimal. Its optimality follows from the fact that the prefix code tree is full, and that the least-frequently occurring symbols are (by constructive definition) at the lowest points on the tree.

The basic static scheme described here is also the theoretical foundation for countless dynamic encoding approaches that calculate the prefix coding on the fly. Some of these even manage to not store the prefix code table at all in the output… instead it's inferred from the output and rebuilt dynamically during decompression.

Data Flow Diagram:



3 Arithmetic coding

3.1 Overview

Huffman codes were a major breakthrough in their time, but they have a flaw: they always encode an output symbol using an integral number of bits. At first glance this may seem obvious and unavoidable, but arithmetic coding showed that this was not the case.

The basic premise of arithmetic coding is to convert the input data stream into a single real number as output. The real number is then represented in binary with the minimum number of digits to ensure it can be accurately decompressed back to the original data.

Arithmetic coding is a form of variable-length entropy encoding used in lossless data compression. Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding such as Huffman coding in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, a fraction *n* where ($0.0 \le n < 1.0$).

3.1.1 Encoding and decoding: overview

In general, each step of the encoding process, except for the very last, is the same; the encoder has basically just three pieces of data to consider:

- The next symbol that needs to be encoded
- The current interval (at the very start of the encoding process, the interval is set to [0,1), but that will change)
- The probabilities the model assigns to each of the various symbols that are possible at this stage (as mentioned earlier, higher-order or adaptive models mean that these probabilities are not necessarily the same in each step.)

The encoder divides the current interval into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol in the current context.

Whichever interval corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.

Example: for the four-symbol model above:

- the interval for NEUTRAL would be [0, 0.6)
- the interval for POSITIVE would be [0.6, 0.8)
- the interval for NEGATIVE would be [0.8, 0.9)
- the interval for END-OF-DATA would be [0.9, 1).

When all symbols have been encoded, the resulting interval unambiguously identifies the sequence of symbols that produced it. Anyone who has the same final interval and model that is being used can reconstruct the symbol sequence that must have entered the encoder to result in that final interval.

It is not necessary to transmit the final interval, however; it is only necessary to transmit *one fraction* that lies within that interval. In particular, it is only necessary to transmit enough digits (in whatever base) of the fraction so that all fractions that begin with those digits fall into the final interval.

3.1.2 Encoding and decoding: example

Consider the process for decoding a message encoded with the given four-symbol model. The message is encoded in the fraction 0.538 (using decimal for clarity, instead of binary; also assuming that there are only as many digits as needed to decode the message.)

The process starts with the same interval used by the encoder: [0,1), and using the same model, dividing it into the same four sub-intervals that the encoder must have. The fraction 0.538 falls into the sub-interval for NEUTRAL, [0, 0.6); this indicates that the first symbol the encoder read must have been NEUTRAL, so this is the first symbol of the message.

Next divide the interval [0, 0.6) into sub-intervals:

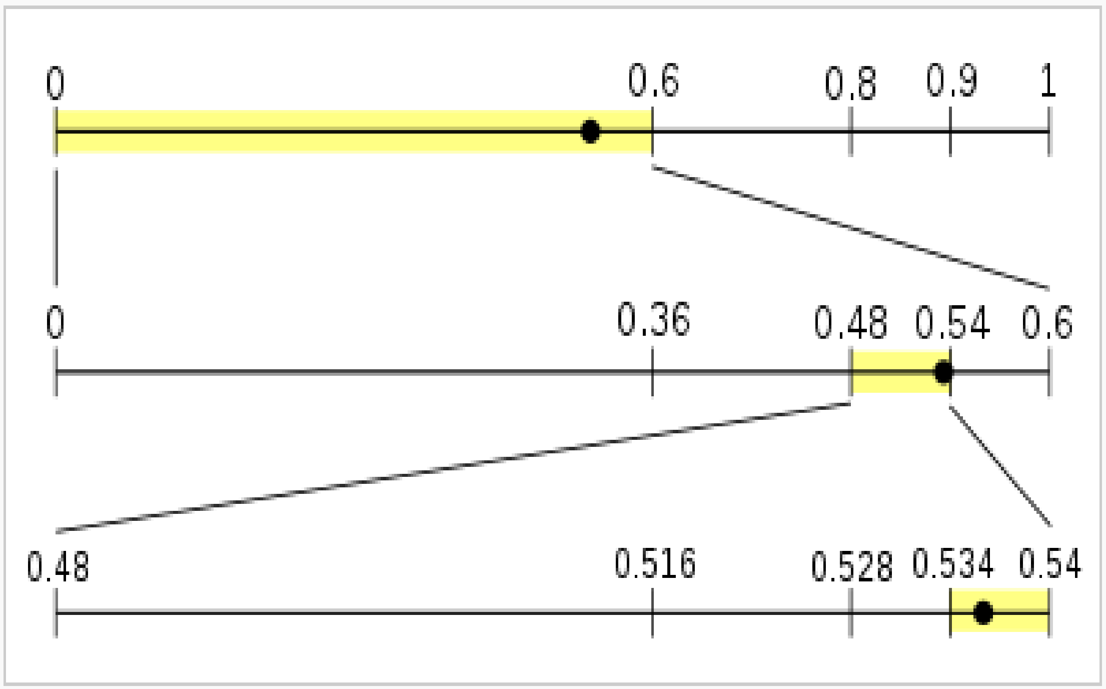
- the interval for NEUTRAL would be [0, 0.36) -- *60% of [0, 0.6)*
- the interval for POSITIVE would be [0.36, 0.48) -- *20% of [0, 0.6)*
- the interval for NEGATIVE would be [0.48, 0.54) -- *10% of [0, 0.6)*
- the interval for END-OF-DATA would be [0.54, 0.6). -- *10% of [0, 0.6)*

Since .538 is within the interval [0.48, 0.54), the second symbol of the message must have been NEGATIVE.

Again divide our current interval into sub-intervals:

- the interval for NEUTRAL would be [0.48, 0.516)
- the interval for POSITIVE would be [0.516, 0.528)
- the interval for NEGATIVE would be [0.528, 0.534)
- the interval for END-OF-DATA would be [0.534, 0.540).

Now .538 falls within the interval of the END-OF-DATA symbol; therefore, this must be the next symbol. Since it is also the internal termination symbol, it means the decoding is complete. If the stream was not internally terminated, there needs to be some other way to indicate where the stream stops. Otherwise, the decoding process could continue forever, mistakenly reading more symbols from the fraction than were in fact encoded into it.



A diagram showing decoding of 0.538 (the circular point) in the example model. The region is divided into subregions proportional to symbol frequencies, then the subregion containing the point is successively subdivided in the same way.

PRINCIPLE OF ARITHMATIC CODING

The entire data set is represented by a single rational number, which is placed between 0 and 1. This range is divided into sub-intervals each representing a certain symbol. The number of sub-intervals is identical to the number of symbols in the current set of symbols and the size is proportional to their probability of appearance. For each symbol in the original data a new interval division takes place, on the basis of the last sub-interval.

Example Sub-Interval:

```
         Probability   Initial  Intervals
a                0.5     (50%)    [0.0; 0.5)
b                0.3     (30%)    [0.5; 0.8)
c                0.2     (20%)    [0.8; 1.0)
Size Current Interval: 1.0
```

```
  Start                'b'                 'a'
a [0.0; 0.5)      [0.50; 0.65) --> [0.500; 0.575)
b [0.5; 0.8) --> [0.65; 0.74)      [0.575; 0.620)
c [0.8; 1.0)      [0.74; 0.80)      [0.620; 0.650)
Size Current Interval:  0.30                0.150
```

Starting with the initial sub-division the symbol 'b' is coded with a number, which is greater or equal than 0.5 and less than 0.8. If the symbol 'b' would stand alone, any number from this interval could be selected.

The encoding of the following symbol bases on the current interval [0.5; 0.8), which will be divided again into sub-intervals. If the second symbol is an 'a', then any number could be coded from the interval [0.50; 0.65). With any further symbol the last current interval have to be divided again and again, the number of significant digits of the code word increases continuously.

3.2 Precise algorithm description

As with Huffman coding, the encoding process starts by counting the occurrences of every input symbol in our input stream. Then we divide the unit segment [0,1] into m separate areas, one for each of the m symbols in our input alphabet. Each symbol receives a specific contiguous sub-range of the unit segment, in direct proportion to its occurrence count, such that the entire unit segment is covered exactly once with no overlap between symbols.

Suppose $X = \{x_i \mid x_i \textit{ is in the input alphabet}\}$, as an ordered set of size m. Let us define n as the total size of our input stream and p(i) as the probability of symbol x_i in the input (number of occurrences of the symbol x_i, divided by n).

We're defining critical points $c(i) = c(i-1) + p(i)$ to divide the unit segment into regions, where c(0)=0 and c(m)=1.

The encoding process can then be described as follows:

1. Initialize lo := 0 and hi := 1.
2. Read one symbol from the input, as x_i.
3. Set lo := lo + (hi-lo) c(i-1)
4. Set hi := lo + (hi-lo) c(i)
5. Repeat from step 2, until we exhaust the input stream
6. Output a floating-point binary encoding of any number between lo and hi, so long is it possesses the minimum number of bits required to indicate it’s within that range.

The decoding process is similar. It performs the above operations in reverse:

1. Take r as the floating-point binary number between 0 and 1 that resulted from a compression output.

2. Determine the minimum value i satisfying c(i) >= r.

3. Output the symbol x_i.

4. Set r := (r – c(i)) / (c(i+1) – c(i))

5. Repeat from step 2, until we exhaust the precision of the original r's bits.

As with Huffman encoding, the pseudo-code above implies that we must include extra data in the compressor's output. The decompressor specifically needs the values c(i), in order to translate back to the original symbols.

3.3 Caveats and implementation details

It should be evident from the pseudo-code that the algorithm as described requires arbitrary-precision floafating point operations. This quickly becomes extremely inefficient; it's silly to manipulate a 10-million digit floating point number when compressing 10 megabytes of data!

Fortunately, we don't need to. Though its concepts are based heavily in real numbers, all practical implementations of arithmetic encoding use some math tricks to implement the algorithm using only integer arithmetic and shift operations.

Full implementations make use of a sliding window of bits over the conceptual arbitrary-precision binary number, shifting bits in and out as necessary. This complicates the implementation of the compressor a bit, and complicates the decompressor tremendously. But this complication pays off, since fixed-precision integer arithmetic is orders of magnitude faster than arbitrary-precision floating-point arithmetic.

The critical point values c(i) are best kept in a flat array, both for the compressor and the decompressor. The compressor stores them indexed on the value i. The decompressor stores them in a reverse-index format, with values of i stored in an array indexed by c(i) values (which are encoded as small integers in the integer-math version).

An important detail glossed over in the pseudo-code is that of marking the end of file. In order for the decompression routine to terminate, it must have some way of determining the condition described in its step 5. There are generally two approaches to this. One is to encode a special end-of-file marker into the compression format. The other is to simply include a bit count in the compression format. Depending on the application (data transmission versus data storage), each method has its uses.

3.4 Real-world optimizations and extensions

Just as the static Huffman coding could be improved by allowing symbol frequencies to change dynamically, so can arithmetic encoding. There are numerous adaptations of arithmetic coding that allow for this. Arithmetic coding is considerably newer than Huffman coding, so there is even more research into dynamic arithmetic coding schemes. All involve modifying the symbol frequencies during the encoding phase, to adapt to a changing data stream. The fanciest methods actually eliminate explicit transmission of symbol frequencies altogether… instead, the decompressor infers them based on the compressed data stream alone.

3.5 Best and worst case performance

Suppose that n is the size of our input stream. We will again ignore m, the number of symbols in the input alphabet. Note that this allows us to ignore the time required to write/read the symbol frequency table, as this gets lost in the limit.

Clearly the initial symbol frequency determination again requires $\Theta(n)$ operations. The compression loop in steps 2-5 then requires $\Theta(n)$ operations as well. Finally, step 6 of compression involves $\Omega(0)$ best case or $O(\log n)$ worst case operations. Thus the compression algorithm as a whole requires $\Theta(n)$ operations. Note that an "operation" here includes a single operation on an arbitrary-precision number on a theoretical arbitrary-precision computer.

If the c(i) values are properly indexed with an efficient data structure, step 2 of the decompression algorithm can be performed in $O(1)$ operations. This means that every step of the decompression algorithm takes constant time, resulting in $\Theta(n)$ performance for decompression.

Now on to compression ratios. In the ideal case, the input consists of a single symbol repeated n times. The ideal compressed data is then actually zero(!) bits to represent the output of the compressor, along with log n bits to indicate the desired length of the original data. So for optimal input, arithmetic coding is capable of compressing n input symbols down to log n output bits!

In the worst case, the input is evenly randomly distributed. In this case the compressed stream is essentially a fixed-size encoding of the input data, resulting in n log m bits. The overhead of log n bits to encode the stream length must be included, for a grand total of n log m + log n bits in the worst case.

3.6 Theoretical significance

The arithmetic encoding scheme is even more optimal than the fabled “optimal” Huffman coding! Its improvements stem from the fact that it can actually utilize fractions of bits per input symbol.

It’s only recently that arithmetic coding has become practical, due to advances in modern hardware. There is a large amount of interest today in various adaptive frequency scheme add-ons to the basic static algorithm, since arithmetic encoding’s accomplishments were a holy grail to decades of researchers.

Comparison of the two algorithms

3.7 Discussion of real-world relative pros and cons

3.7.1 Time performance

Though both algorithms have similar best/worst runtime characteristics when expressed in Big-O notation, the actual runtimes can vary quite a bit. Arithmetic coding requires many more actual operations per iteration than Huffman. In fact, the fully floating-point pseudo-code supplied above will actually require O(n) atomic operations for each arbitrary-precision floating-point operation.

Thus the naïve arithmetic algorithm is an order of magnitude slower than Huffman. However, even the optimized integer arithmetic version generally performs some constant factor slower than Huffman coding.

3.7.2 Space requirements

Both algorithms stream their compressors and decompressors. Both algorithms require O(m log m) storage for frequency data during compression. Time-efficient implementations of both algorithms require O(m) storage during decompression.

Since the cardinality of the input alphabet is generally small, all of the above storage requirements are quite minimal in practice. They're essentially ignorable in all but the most severely constrained applications.

3.7.3 Relative ease/difficulty of implementations

Huffman coding is heavily used everywhere for a very good reason: it's easy! Huffman coding is extremely straightforward to implement, and requires little processing power. Both the compressor and decompressor can be implemented in relatively few lines of code, and can easily be adapted to the simplest of processors. Additionally, the simplicity of the algorithm aides quality assurance and, in the event of a problem, debugging.

The decompressor in particular is completely trivial, requiring little in the way of data structures.

Arithmetic coding on the other hand, is much more complicated. The pseudo-code supplied above is "magical" enough… but consider what happens when it's further obfuscated by integer arithmetic! Even after a programmer finishes implementation, the debugging of an arithmetic coding program gone wrong is an intimidating proposition. The state of an arithmetic compressor and decompressor are near-gibberish to a human.

3.8 Survey of actual compression ratios for sample data

To collect the following data, I took simple implementations of both algorithms and applied them to obvious sample data. While certainly not an exhaustive survey, it does give some simple real-world data.

Each compressed file size is expressed as a percentage of the uncompressed file size.

Description	Original Size	Huffman	Arithmetic
Large file of constant bits A file filled with binary 0s.	1024 KB	0.025%	0.041%
Randomly generated binary data Created with a C program.	1024 KB	100.024%	100.480%
Block of English text Shakespeare’s Romeo and Juliet.	156 KB	60.11%	59.74%
Programming source code Large java source file.	63 KB	57.30%	56.90%
Executable machine code A Linux executable file.	2910 KB	69.83%	68.95%

It is interesting to note that the arithmetic coder has a better theoretical best case compression ratio, but in practice (for these particular implementations) this is not realized. Indeed, it seems that the Huffman implementation out-performs the arithmetic one for both degenerate cases. However, it’s by only a slim margin, and such cases are usually the exception and not the rule.For English text, programming source code, and executable code, the implementations performed exactly how one would naïvely expect: the arithmetic implementation shaves a small (but noticeable) amount off of Huffman’s. This is expected since the arithmetic algorithm really does just still those extra fractional bits.

4 JPEG Compression

One of the hottest topics in image compression technology today is JPEG. The acronym JPEG stands for the Joint Photographic Experts Group, a standards committee that had its origins within the International Standard Organization (ISO). In 1982, the ISO formed the Photographic Experts Group (PEG) to research methods of transmitting video, still images, and text over ISDN (Integrated Services Digital Network) lines. PEG's goal was to produce a set of industry standards for the transmission of graphics and image data over digital communications networks.

In 1986, a subgroup of the CCITT began to research methods of compressing color and gray-scale data for facsimile transmission. The compression methods needed for color facsimile systems were very similar to those being researched by PEG. It was therefore agreed that the two groups should combine their resources and work together toward a single standard.

In 1987, the ISO and CCITT combined their two groups into a joint committee that would research and produce a single standard of image data compression for both organizations to use. This new committee was JPEG.

Although the creators of JPEG might have envisioned a multitude of commercial applications for JPEG technology, a consumer public made hungry by the marketing promises of imaging and multimedia technology are benefiting greatly as well. Most previously developed compression methods do a relatively poor job of compressing continuous-tone image data; that is, images containing hundreds or thousands of colors taken from real-world subjects. And very few file formats can support 24-bit raster images.

GIF, for example, can store only images with a maximum pixel depth of eight bits, for a maximum of 256 colors. And its LZW compression algorithm does not work very well on typical scanned image data. The low-level noise commonly found in such data defeats LZW's ability to recognize repeated patterns.

Both TIFF and BMP are capable of storing 24-bit data, but in their pre-JPEG versions are capable of using only encoding schemes (LZW and RLE, respectively) that do not compress this type of image data very well.

JPEG provides a compression method that is capable of compressing continuous-tone image data with a pixel depth of 6 to 24 bits with reasonable speed and efficiency. And although JPEG itself does not define a standard image file format, several have been invented or modified to fill the needs of JPEG data storage.

4.1.1 JPEG in Perspective

Unlike all of the other compression methods described so far in this chapter, JPEG is not a single algorithm. Instead, it may be thought of as a toolkit of image compression methods that may be altered to fit the needs of the user. JPEG may be adjusted to produce very small, compressed images that are of relatively poor quality in appearance but still suitable for many applications. Conversely, JPEG is capable of producing very high-quality compressed images that are still far smaller than the original uncompressed data.

JPEG is also different in that it is primarily a lossy method of compression. Most popular image format compression schemes, such as RLE, LZW, or the CCITT standards, are lossless compression methods. That is, they do not discard any data during the encoding process. An image compressed using a lossless method is guaranteed to be identical to the original image when uncompressed.

Lossy schemes, on the other hand, throw useless data away during encoding. This is, in fact, how lossy schemes manage to obtain superior compression ratios over most lossless schemes. JPEG was designed specifically to discard information that the human eye cannot easily see. Slight changes in color are not perceived well by the human eye, while slight changes in intensity (light and dark) are. Therefore JPEG's lossy encoding tends to be more frugal with the gray-scale part of an image and to be more frivolous with the color.

JPEG was designed to compress color or gray-scale continuous-tone images of real-world subjects: photographs, video stills, or any complex graphics that resemble natural subjects. Animations, ray tracing, line art, black-and-white documents, and typical vector graphics don't compress very well under JPEG and shouldn't be expected to. And, although JPEG is now used to provide motion video compression, the standard makes no special provision for such an application.

The fact that JPEG is lossy and works only on a select type of image data might make you ask, "Why bother to use it?" It depends upon your needs. JPEG is an excellent way to store 24-bit photographic images, such as those used in imaging and multimedia applications. JPEG 24-bit (16 million color) images are superior in appearance to 8-bit (256 color) images on a VGA display and are at their most spectacular when using 24-bit display hardware (which is now quite inexpensive).

The amount of compression achieved depends upon the content of the image data. A typical photographic-quality image may be compressed from 20:1 to 25:1 without experiencing any noticeable degradation in quality. Higher compression ratios will result in image files that differ noticeably from the original image but still have an overall good image quality. And achieving a 20:1 or better compression ratio in many cases not only saves disk space, but also reduces transmission time across data networks and phone lines.

An end user can "tune" the quality of a JPEG encoder using a parameter sometimes called a *quality setting* or a *Q factor*. Although different implementations have varying scales of Q factors, a range of 1 to 100 is typical. A factor of 1 produces the smallest, worst quality images; a factor of 100 produces the largest, best quality images. The optimal Q factor depends on the image content and is therefore different for every image. The art of JPEG compression is finding the lowest Q factor that produces an image that is visibly acceptable, and preferably as close to the original as possible.

The JPEG library supplied by the Independent JPEG Group uses a quality setting scale of 1 to 100. To find the optimal compression for an image using the JPEG library, follow these steps:

1. Encode the image using a quality setting of 75 .
2. If you observe unacceptable defects in the image, increase the value, and re-encode the image.
3. If the image quality is acceptable, decrease the setting until the image quality is barely acceptable. This will be the optimal quality setting for this image.
4. Repeat this process for every image you have (or just encode them all using a quality setting of 75).

JPEG isn't always an ideal compression solution. There are several reasons:

- As we have said, JPEG doesn't fit every compression need. Images containing large areas of a single color do not compress very well. In fact, JPEG will introduce "artifacts" into such images that are visible against a flat background, making them considerably worse in appearance than if you used a conventional lossless compression method. Images of a "busier" composition contain even worse artifacts, but they are considerably less noticeable against the image's more complex background.
- JPEG can be rather slow when it is implemented only in software. If fast decompression is required, a hardware-based JPEG solution is your best bet, unless you are willing to wait for a faster software-only solution to come along or buy a faster computer.
- JPEG is not trivial to implement. It is not likely you will be able to sit down and write your own JPEG encoder/decoder in a few evenings. We recommend that you obtain a third-party JPEG library, rather than writing your own.
- JPEG is not supported by very many file formats. The formats that do support JPEG are all fairly new and can be expected to be revised at frequent intervals.

4.1.2 Baseline JPEG

The JPEG specification defines a minimal subset of the standard called baseline JPEG, which all JPEG-aware applications are required to support. This baseline uses an encoding scheme based on the Discrete Cosine Transform (DCT) to achieve compression. DCT is a generic name for a class of operations identified and published some years ago. DCT-based algorithms have since made their way into various compression methods.

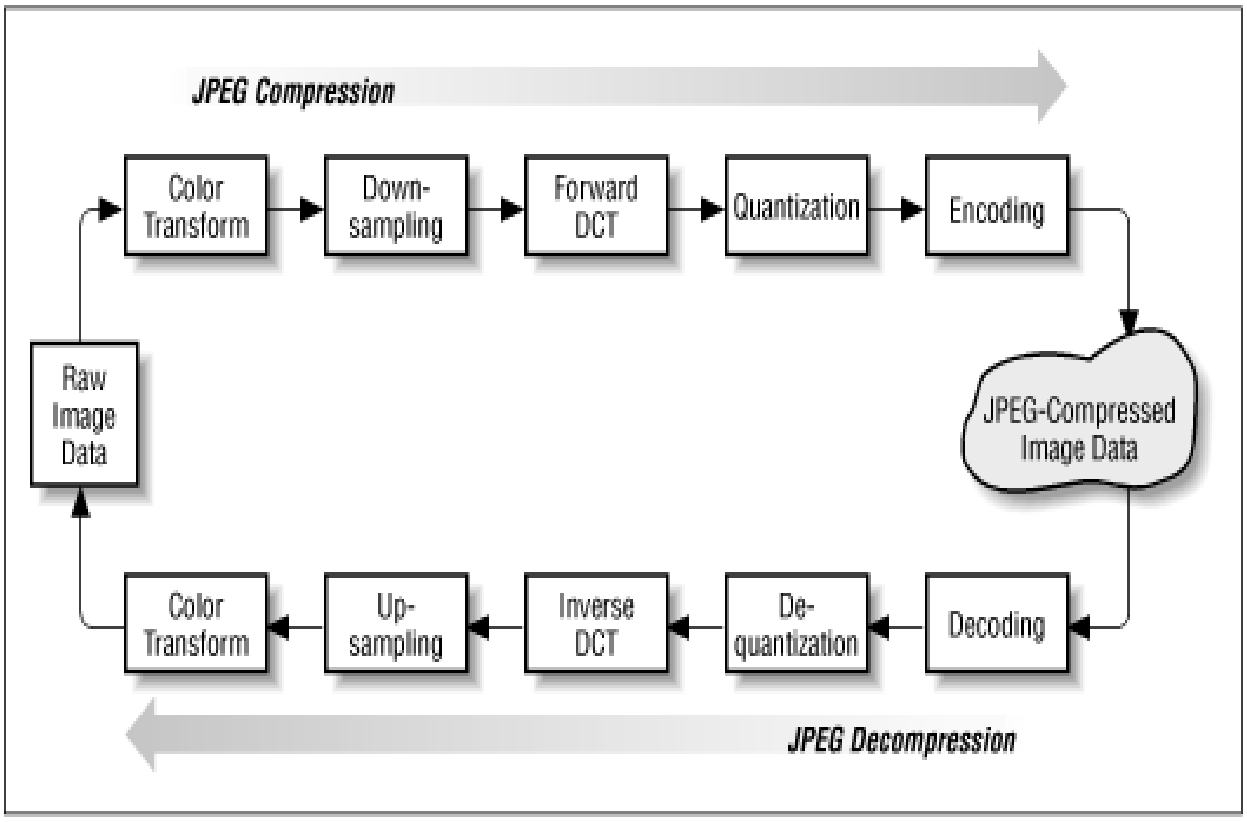
DCT-based encoding algorithms are always lossy by nature. DCT algorithms are capable of achieving a high degree of compression with only minimal loss of data. This scheme is effective only for compressing continuous-tone images in which the differences between adjacent pixels are usually small. In practice, JPEG works well only on images with depths of at least four or five bits per color channel. The baseline standard actually specifies eight bits per input sample. Data of lesser bit depth can be handled by scaling it up to eight bits per sample, but the results will be bad for low-bit-depth source data, because of the large jumps between adjacent pixel values. For similar reasons, colormapped source data does not work very well, especially if the image has been dithered.

The JPEG compression scheme is divided into the following stages:

1. Transform the image into an optimal color space.
2. Downsample chrominance components by averaging groups of pixels together.
3. Apply a Discrete Cosine Transform (DCT) to blocks of pixels, thus removing redundant image data.
4. Quantize each block of DCT coefficients using weighting functions optimized for the human eye.
5. Encode the resulting coefficients (image data) using a Huffman variable word-length algorithm to remove redundancies in the coefficients.

Figure: summarizes these steps, and the following subsections look at each of them in turn. Note that JPEG decoding performs the reverse of these steps.

4.1.2.1 Figure : JPEG compression and decompression



Transform the image

The JPEG algorithm is capable of encoding images that use any type of color space. JPEG itself encodes each component in a color model separately, and it is completely independent of any color-space model, such as RGB, HSI, or CMY. The best compression ratios result if a luminance/chrominance color space, such as YUV or YCbCr, is used.

Most of the visual information to which human eyes are most sensitive is found in the high-frequency, gray-scale, luminance component (Y) of the YCbCr color space. The other two chrominance components (Cb and Cr) contain high-frequency color information to which the human eye is less sensitive. Most of this information can therefore be discarded.

In comparison, the RGB, HSI, and CMY color models spread their useful visual image information evenly across each of their three color components, making the selective discarding of information very difficult. All three color components would need to be encoded at the highest quality, resulting in a poorer compression ratio. Gray-scale images do not have a color space as such and therefore do not require transforming.

4.1.3 Downsample chrominance components

The simplest way of exploiting the eye's lesser sensitivity to chrominance information is simply to use fewer pixels for the chrominance channels. For example, in an image nominally 1000x1000 pixels, we might use a full 1000x1000 luminance pixels but only 500x500 pixels for each chrominance component. In this representation, each chrominance pixel covers the same area as a 2x2 block of luminance pixels. We store a total of six pixel values for each 2x2 block (four luminance values, one each for the two chrominance channels), rather than the twelve values needed if each component is represented at full resolution. Remarkably, this 50 percent reduction in data volume has almost no effect on the perceived quality of most images. Equivalent savings are not possible with conventional color models such as RGB, because in RGB each color channel carries some luminance information and so any loss of resolution is quite visible.

When the uncompressed data is supplied in a conventional format (equal resolution for all channels), a JPEG compressor must reduce the resolution of the chrominance channels by *downsampling*, or averaging together groups of pixels. The JPEG standard allows several different choices for the sampling ratios, or relative sizes, of the downsampled channels. The luminance channel is always left at full resolution (1:1 sampling). Typically both chrominance channels are downsampled 2:1 horizontally and either 1:1 or 2:1 vertically, meaning that a chrominance pixel covers the same area as either a 2x1 or a 2x2 block of luminance pixels. JPEG refers to these downsampling processes as 2h1v and 2h2v sampling, respectively.

Another notation commonly used is 4:2:2 sampling for 2h1v and 4:2:0 sampling for 2h2v; this notation derives from television customs (color transformation and downsampling have been in use since the beginning of color TV transmission). 2h1v sampling is fairly common because it corresponds to National Television Standards Committee (NTSC) standard TV practice, but it offers less compression than 2h2v sampling, with hardly any gain in perceived quality.

4.1.4 Discrete Cosine Transform

The image data is divided up into 8x8 blocks of pixels. (From this point on, each color component is processed independently, so a "pixel" means a single value, even in a color image.) A DCT is applied to each 8x8 block. DCT converts the spatial image representation into a frequency map: the low-order or "DC" term represents the average value in the block, while successive higher-order ("AC") terms represent the strength of more and more rapid changes across the width or height of the block. The highest AC term represents the strength of a cosine wave alternating from maximum to minimum at adjacent pixels.

The DCT calculation is fairly complex; in fact, this is the most costly step in JPEG compression. The point of doing it is that we have now separated out the high- and low-frequency information present in the image. We can discard high-frequency data easily without losing low-frequency information. The DCT step itself is lossless except for roundoff errors.

4.1.5 Quantize each block

To discard an appropriate amount of information, the compressor divides each DCT output value by a "quantization coefficient" and rounds the result to an integer. The larger the quantization coefficient, the more data is lost, because the actual DCT value is represented less and less accurately. Each of the 64 positions of the DCT output block has its own quantization coefficient, with the higher-order terms being quantized more heavily than the low-order terms (that is, the higher-order terms have larger quantization coefficients). Furthermore, separate quantization tables are employed for luminance and chrominance data, with the chrominance data being quantized more heavily than the luminance data. This allows JPEG to exploit further the eye's differing sensitivity to luminance and chrominance.

It is this step that is controlled by the "quality" setting of most JPEG compressors. The compressor starts from a built-in table that is appropriate for a medium-quality setting and increases or decreases the value of each table entry in inverse proportion to the requested quality. The complete quantization tables actually used are recorded in the compressed file so that the decompressor will know how to (approximately) reconstruct the DCT coefficients.

Selection of an appropriate quantization table is something of a black art. Most existing compressors start from a sample table developed by the ISO JPEG committee. It is likely that future research will yield better tables that provide more compression for the same perceived image quality. Implementation of improved tables should not cause any compatibility problems, because decompressors merely read the tables from the compressed file; they don't care how the table was picked.

4.1.6 Encode the resulting coefficients

The resulting coefficients contain a significant amount of redundant data. Huffman compression will losslessly remove the redundancies, resulting in smaller JPEG data. An optional extension to the JPEG specification allows arithmetic encoding to be used instead of Huffman for an even greater compression ratio. At this point, the JPEG data stream is ready to be transmitted across a communications channel or encapsulated inside an image file format.

4.1.7 JPEG Extensions (Part 1)

What we have examined thus far is only the baseline specification for JPEG. A number of extensions have been defined in Part 1 of the JPEG specification that provide progressive image buildup, improved compression ratios using arithmetic encoding, and a lossless compression scheme. These features are beyond the needs of most JPEG implementations and have therefore been defined as "not required to be supported" extensions to the JPEG standard.

4.1.8 Progressive image buildup

Progressive image buildup is an extension for use in applications that need to receive JPEG data streams and display them on the fly. A baseline JPEG image can be displayed only after all of the image data has been received and decoded. But some applications require that the image be displayed after only some of the data is received. Using a conventional compression method, this means displaying the first few scan lines of the image as it is decoded. In this case, even if the scan lines were interlaced, you would need at least 50 percent of the image data to get a good clue as to the content of the image. The progressive buildup extension of JPEG offers a better solution.

Progressive buildup allows an image to be sent in layers rather than scan lines. But instead of transmitting each bitplane or color channel in sequence (which wouldn't be very useful), a succession of images built up from approximations of the original image are sent. The first scan provides a low-accuracy representation of the entire image--in effect, a very low-quality JPEG compressed image. Subsequent scans gradually refine the image by increasing the effective quality factor. If the data is displayed on the fly, you would first see a crude, but recognizable, rendering of the whole image. This would appear very quickly because only a small amount of data would need to be transmitted to produce it. Each subsequent scan would improve the displayed image's quality one block at a time.

A limitation of progressive JPEG is that each scan takes essentially a full JPEG decompression cycle to display. Therefore, with typical data transmission rates, a very fast JPEG decoder (probably specialized hardware) would be needed to make effective use of progressive transmission.

A related JPEG extension provides for hierarchical storage of the same image at multiple resolutions. For example, an image might be stored at 250x250, 500x500, 1000x1000, and 2000x2000 pixels, so that the same image file could support display on low-resolution screens, medium-resolution laser printers, and high-resolution imagesetters. The higher-resolution images are stored as differences from the lower-resolution ones, so they need less space than they would need if they were stored independently. This is not the same as a progressive series, because each image is available in its own right at the full desired quality.

4.1.9 Arithmetic encoding

The baseline JPEG standard defines Huffman compression as the final step in the encoding process. A JPEG extension replaces the Huffman engine with a binary arithmetic entropy encoder. The use of an arithmetic coder reduces the resulting size of the JPEG data by a further 10 percent to 15 percent over the results that would be achieved by the Huffman coder. With no change in resulting image quality, this gain could be of importance in implementations where enormous quantities of JPEG images are archived.

Arithmetic encoding has several drawbacks:

- Not all JPEG decoders support arithmetic decoding. Baseline JPEG decoders are required to support only the Huffman algorithm.
- The arithmetic algorithm is slower in both encoding and decoding than Huffman.
- The arithmetic coder used by JPEG (called a *Q-coder*) is owned by IBM and AT&T. (Mitsubishi also holds patents on arithmetic coding.) You must obtain a license from the appropriate vendors if their Q-coders are to be used as the back end of your JPEG implementation.

4.1.10 Lossless JPEG compression

A question that commonly arises is "At what Q factor does JPEG become lossless?" The answer is "never." Baseline JPEG is a lossy method of compression regardless of adjustments you may make in the parameters. In fact, DCT-based encoders are always lossy, because roundoff errors are inevitable in the color conversion and DCT steps. You can suppress deliberate information loss in the downsampling and quantization steps, but you still won't get an exact recreation of the original bits. Further, this minimum-loss setting is a very inefficient way to use lossy JPEG.

The JPEG standard does offer a separate lossless mode. This mode has nothing in common with the regular DCT-based algorithms, and it is currently implemented only in a few commercial applications. JPEG lossless is a form of Predictive Lossless Coding using a 2D Differential Pulse Code Modulation (DPCM) scheme. The basic premise is that the value of a pixel is combined with the values of up to three neighboring pixels to form a predictor value. The predictor value is then subtracted from the original pixel value. When the entire bitmap has been processed, the resulting predictors are compressed using either the Huffman or the binary arithmetic entropy encoding methods described in the JPEG standard.

Lossless JPEG works on images with 2 to 16 bits per pixel, but performs best on images with 6 or more bits per pixel. For such images, the typical compression ratio achieved is 2:1. For image data with fewer bits per pixels, other compression schemes do perform better.

4.1.11 JPEG Extensions (Part 2)

The following JPEG extensions are described in Part 2 of the JPEG specification.

4.1.12 Variable quantization

Variable quantization is an enhancement available to the quantization procedure of DCT-based processes. This enhancement may be used with any of the DCT-based processes defined by JPEG with the exception of the baseline process.

The process of quantization used in JPEG quantizes each of the 64 DCT coefficients using a corresponding value from a quantization table. Quantization values may be redefined prior to the start of a scan but must not be changed once they are within a scan of the compressed data stream.

Variable quantization allows the scaling of quantization values within the compressed data stream. At the start of each 8x8 block is a quantizer scale factor used to scale the quantization table values within an image component and to match these values with the AC coefficients stored in the compressed data. Quantization values may then be located and changed as needed.

Variable quantization allows the characteristics of an image to be changed to control the quality of the output based on a given model. The variable quantizer can constantly adjust during decoding to provide optimal output.

The amount of output data can also be decreased or increased by raising or lowering the quantizer scale factor. The maximum size of the resulting JPEG file or data stream may be imposed by constant adaptive adjustments made by the variable quantizer.

The variable quantization extension also allows JPEG to store image data originally encoded using a variable quantization scheme, such as MPEG. For MPEG data to be accurately transcoded into another format, the other format must support variable quantization to maintain a high compression ratio. This extension allows JPEG to support a data stream originally derived from a variably quantized source, such as an MPEG I-frame.

4.1.13 Selective refinement

Selective refinement is used to select a region of an image for further enhancement. This enhancement improves the resolution and detail of a region of an image. JPEG supports three types of selective refinement: hierarchical, progressive, and component. Each of these refinement processes differs in its application, effectiveness, complexity, and amount of memory required.

- Hierarchical selective refinement is used only in the hierarchical mode of operation. It allows for a region of a frame to be refined by the next differential frame of a hierarchical sequence.
- Progressive selective refinement is used only in the progressive mode and adds refinement. It allows a greater bit resolution of zero and non-zero DCT coefficients in a coded region of a frame.
- Component selective refinement may be used in any mode of operation. It allows a region of a frame to contain fewer colors than are defined in the frame header.

4.1.14 Image tiling

Tiling is used to divide a single image into two or more smaller subimages. Tiling allows easier buffering of the image data in memory, quicker random access of the image data on disk, and the storage of images larger than 64Kx64K samples in size. JPEG supports three types of tiling: simple, pyramidal, and composite.

- Simple tiling divides an image into two or more fixed-size tiles. All simple tiles are coded from left to right and from top to bottom and are contiguous and non-overlapping. All tiles must have the same number of samples and component identifiers and must be encoded using the same processes. Tiles on the bottom and right of the image may be smaller than the designated size of the image dimensions and will therefore not be a multiple of the tile size.

- Pyramidal tiling also divides the image into tiles, but each tile is also tiled using several different levels of resolution. The model of this process is the JPEG Tiled Image Pyramid (JTIP), which is a model of how to create a multi-resolution pyramidal JPEG image.

  A JTIP image stores successive layers of the same image at different resolutions. The first image stored at the top of the pyramid is one-sixteenth of the defined screen size and is called a *vignette*. This image is used for quick displays of image contents, especially for file browsers. The next image occupies one-fourth of the screen and is called an *imagette*. This image is typically used when two or more images must be displayed at the same time on the screen. The next is a low-resolution, full-screen image, followed by successively higher-resolution images and ending with the original image.

  Pyramidal tiling typically uses the process of "internal tiling," where each tile is encoded as part of the same JPEG data stream. Tiles may optionally use the process of "external tiling," where each tile is a separately encoded JPEG data stream. External tiling may allow quicker access of image data, easier application of image encryption, and enhanced compatibility with certain JPEG decoders.

- Composite tiling allows multiple-resolution versions of images to be stored and displayed as a *mosaic*. Composite tiling allows overlapping tiles that may be different sizes and have different scaling factors and compression parameters. Each tile is encoded separately and may be combined with other tiles without resampling.

4.1.15 SPIFF (Still Picture Interchange File Format)

SPIFF is an officially sanctioned JPEG file format that is intended to replace the defacto JFIF (JPEG File Interchange Format) format in use today. SPIFF includes all of the features of JFIF and adds quite a bit more functionality. SPIFF is designed so that properly written JFIF readers will read SPIFF-JPEG files as well.

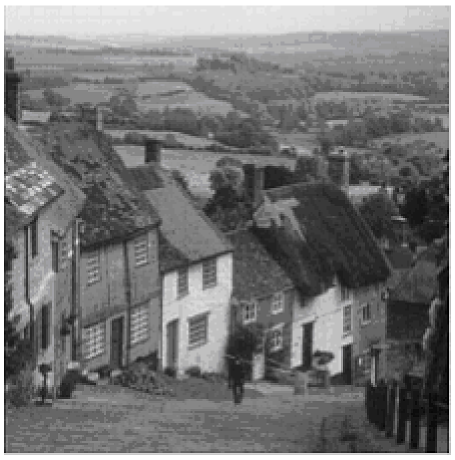
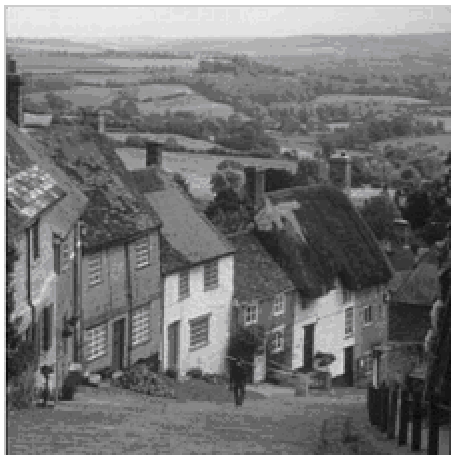
For more information, see the article about SPIFF.

4.1.16 Other extensions

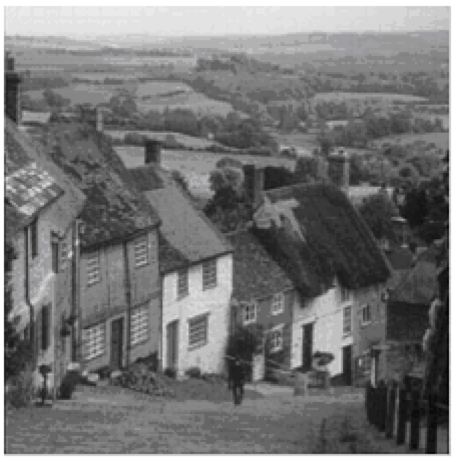
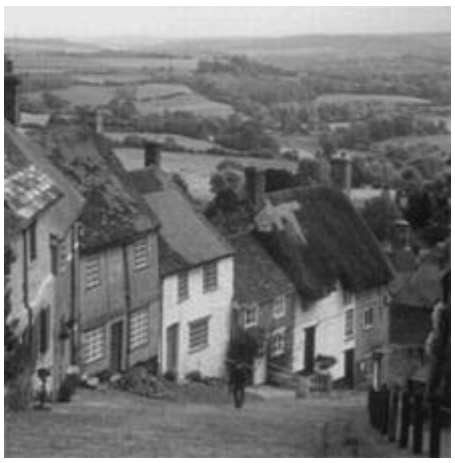
Other JPEG extensions include the addition of a version marker segment that stores the minimum level of functionality required to decode the JPEG data stream. Multiple version markers may be included to mark areas of the data stream that have differing minimum functionality requirements. The version marker also contains information indicating the processes and extension used to encode the JPEG data stream.

5 RESULT AND ANALYSIS

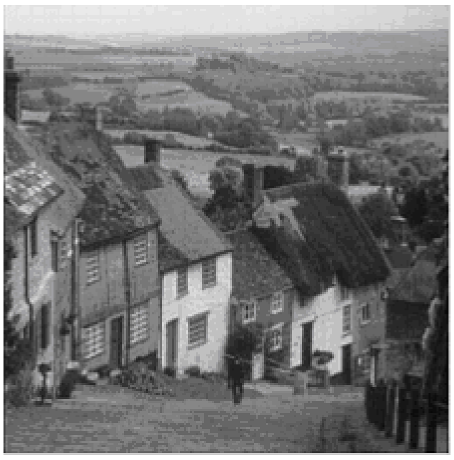
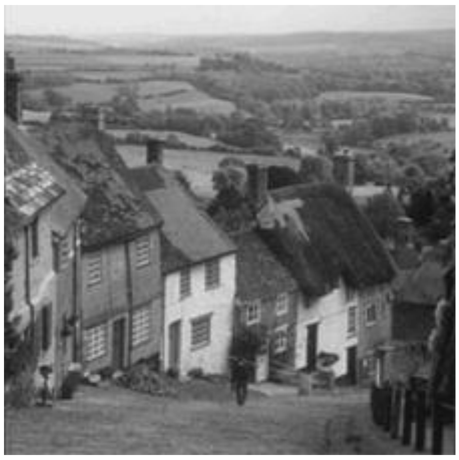
Different numbers of layer of the MLP were tested. The result showed that a three level model is enough, as the PSNR and the quality do not improve much in adding more level.

	
Structure: 15-5-6	15-5-5-5-6

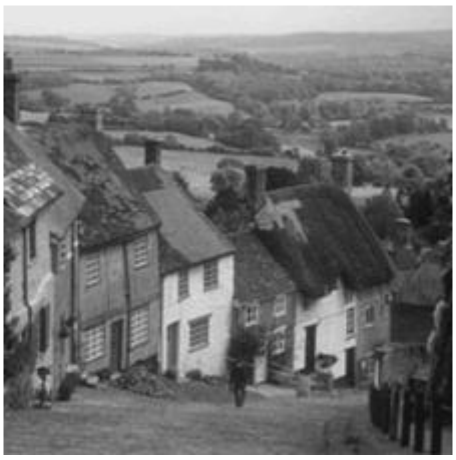
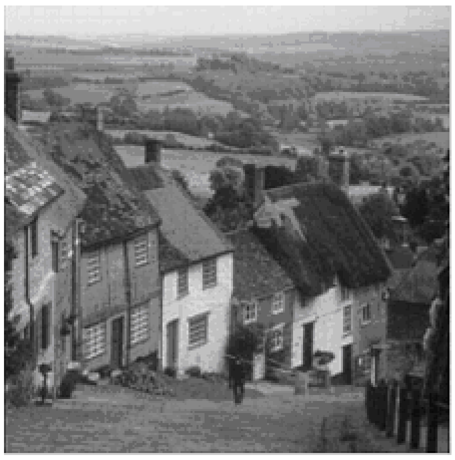
Different numbers of hidden nodes were tested. The result showed that 5 hidden nodes are enough to cheat the human eyes.

	
Structure: 15-5-6	Structure: 15-8-6

Different learning rates were tested. The result showed that 0.01 is sufficient.

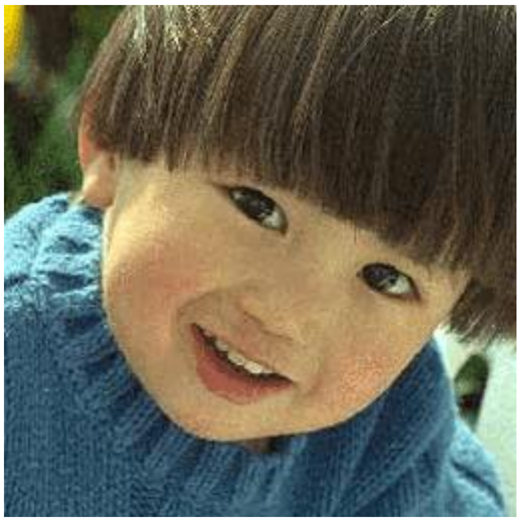
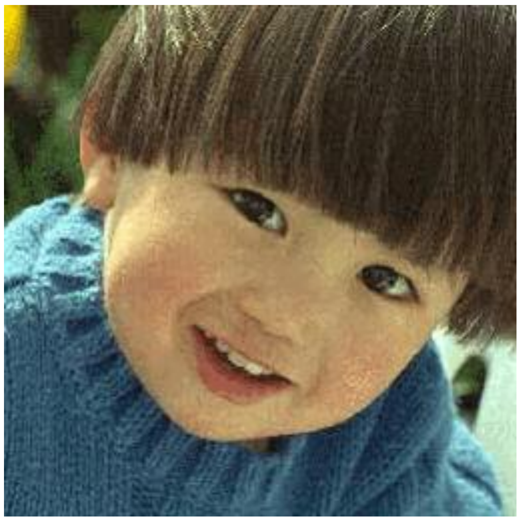
	
Learning rate = 0.01	Learning rate = 0.05

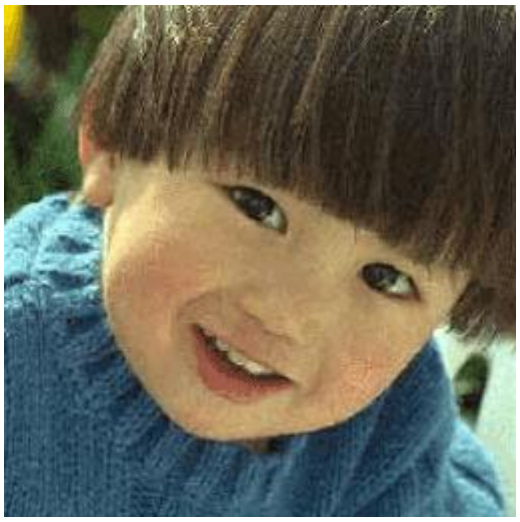
Different momentums were applied. The result showed momentum = 0.5 yield the best result.

	
Momentum = 0.7	Momentum=0

Training sets were extracted from different images. One set of training data was extracted from each image. Then, they were fed to the MLP with the optimal structure obtained above. Comparison is made by the PSNR and the image quality from human eyes.

<u>Experiment #1: color image (train and test with the same image)</u>

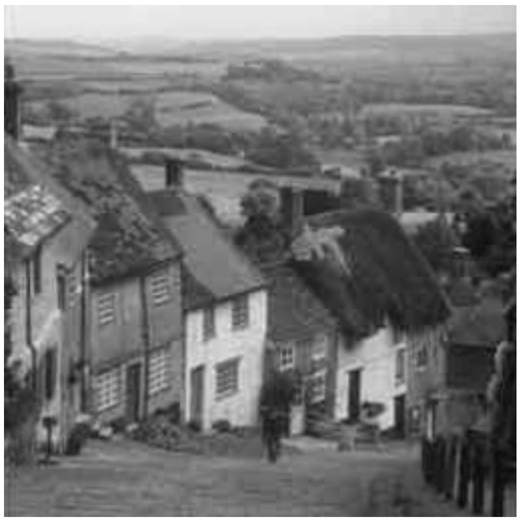
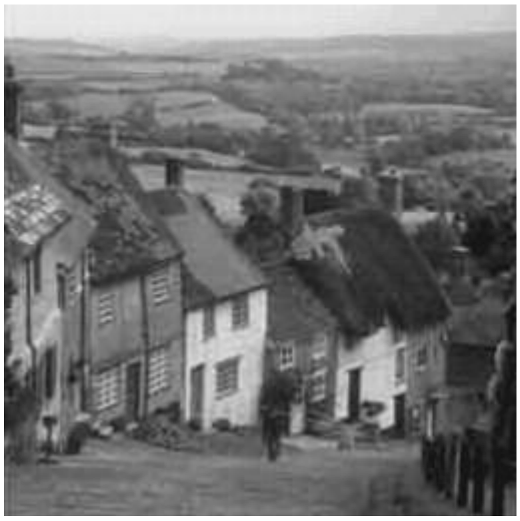
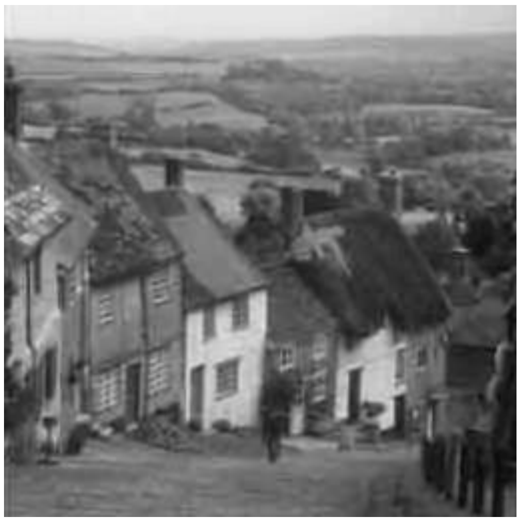
JPEG (0.18 bpp) PSNR = 38.2464 (dB)	
MLP postprocessed PSNR = 37.9718 (dB)	

Polynomial Fitting PSNR = 37.6817 (dB)	

The blocking artifacts decrease a little in the MLP image. However, the PSNR also decrease in the MLP image. It proves that the evaluation of the image quality is different in PSNR and human eyes.

<u>Experiment #2: grayscale image (train with a high bpp image, test with a low bpp image)</u>

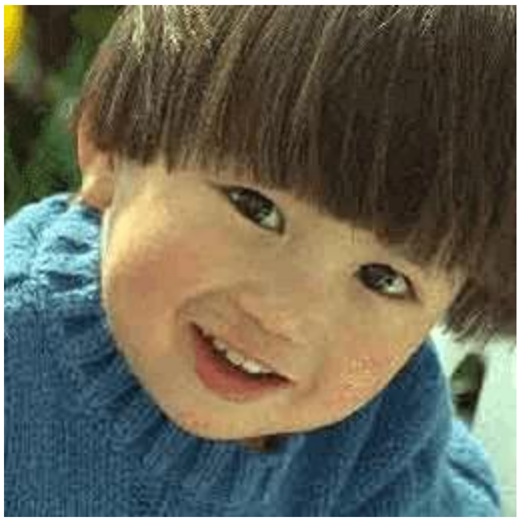
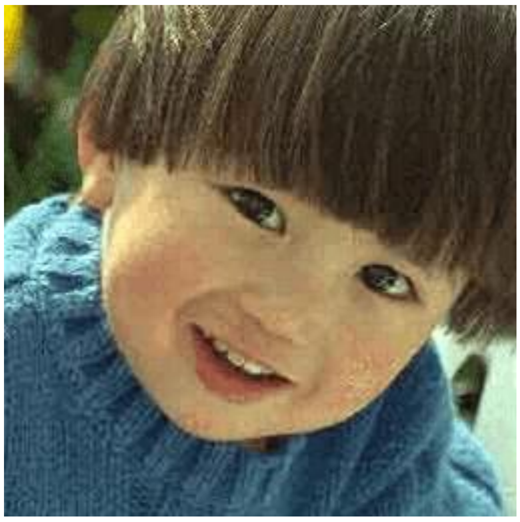
Training JPEG bit rate = 0.255 bpp

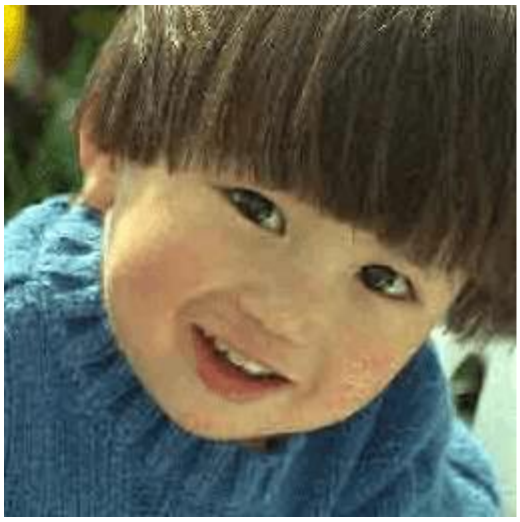
JPEG (0.085 bpp) PSNR = 39.5696 (dB)	
MLP postprocessed PSNR = 39.6552 (dB)	
Polynomial Fitting PSNR = 39.2868 (dB)	

Blocking artifact decreases by the MLP postprocess from human eyes. PSNR also increases after MLP postprocess. Polynomial fitting decreases blocking artifact also but it blurs the image which lead to the decrease in PSNR.

Experiment #4: color image (train with a high bpp image, test with a low bpp image)

Training JPEG image bit rate = 0.374 bpp

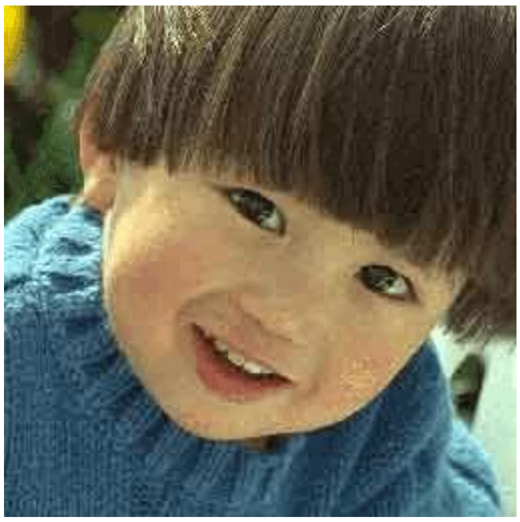
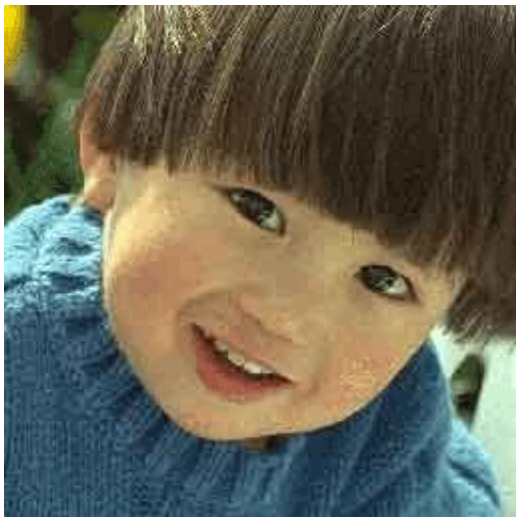
JPEG (0.065 bpp) PSNR = 37.4064 (dB)	
MLP postprocessed PSNR = 37.3664 (dB)	

Polynomial Fitting PSNR = 37.1856 (dB)	

The case of color image is not as good as the one in grayscale image. Both the human eyes quality and the PSNR decreased after MLP postprocessing. On the other hand, polynomial fitting did a good job in decreasing the block artifact, yet the PSNR still decreases.

Experiment #5: train with a high bpp grayscale image, test with a low bpp color image

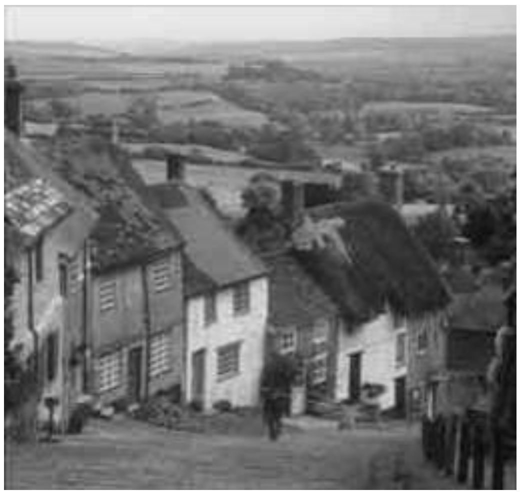
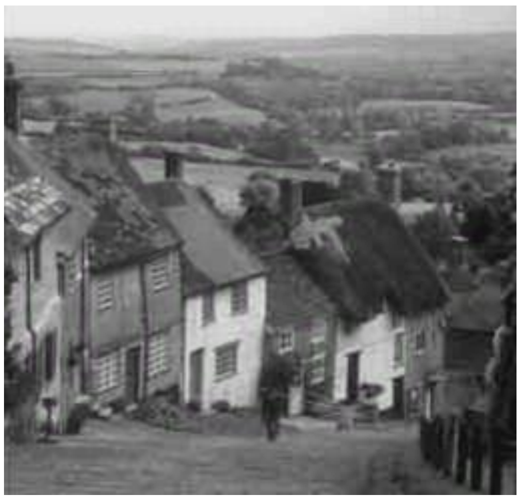
Training JPEG image bit rate = 0.255 bpp

JPEG (0.065 bpp) PSNR = 37.4064 (dB)	
MLP postprocessed PSNR = 37.4312 (dB)	

Although PSNR is increased in the MLP postprocessed image, it is obvious that the blocking artifacts are still there.

Experiment #6: train with a high bpp color image, test with a low bpp grayscale image

Training JPEG image bit rate = 0.374 bpp

JPEG (0.085 bpp) PSNR = 39.5696 (dB)	
MLP postprocessed PSNR = 39.125 (dB)	

Not even blocking artifact was not improved; the MLP postprocess introduced more noise to the image making the PSNR decreases. Moreover, as the MLP was trained by color image, there was some color shift in the MLP postprocessed image.

Assessment of Results

The images, and the decompressed images may be seen in Appendix C. All of the programs were run from the hard disk of a 40MHz 486DX PC. The times quoted are machine dependent and also would be slower if run from a floppy disk. It should be noted that much higher compression ratios would be attainable with grayscale or colour images, as the original black and white images are already down to one bit per pixel. However, in absolute terms, the compressed files are remarkably small for images; with man.bmp compressed to a size where each byte represents over 50 pixels, or over 6 pixels per bit. The image lady.bmp actually increased in size after the fractal compression, due to the relatively low amount of all white areas, and hence a larger proportion of the image needed to be covered by domain blocks. Correspondingly, man.bmp achieved the highest compression ratio, due to the large quantity of plain white space. Compression ratios may be improved by increasing the size of the domain blocks, but this would be at the expense of the quality of the decompressed image. Compression time is obviously related to the size of the initial file, but is also effected by the nature of the image.

Even quite small errors in black and white cartoon style images can be unacceptable, so it is not surprising that the images look quite distorted. The uniform 'grey' areas in particular were poorly reproduced, as the search for a range block which, after a contractive transformation had a similar pixel pattern, was unsuccessful. The quality of the images could be improved by enlarging the range pool, i.e. allowing more possible range blocks, e.g. one beginning every two rather than every four pixels. Alternatively, a larger number of contractive transformations could be made available. Both of these possible solutions would be at the expense of compression time, but would not necessarily decrease the compression ratio. Errors would be harder to detect in grayscale or colour images. The lowest error was attained with the image breeze.bmp, due to a higher than average amount of affine redundancy, in this case all black and all white areas, with fewer blocks of pixels containing fine detail.

To conclude, neither the compression ratios, nor the quality of the images is particularly impressive, however, a sensible balance between the two was reached. The program was written for clarity, and to serve as an example of the method, rather than optimized for speed or image quality. In this respect, it was a success. A program dealing with grayscale or colour images would have achieved better compression ratios and deteriorations in quality would be harder to detect.

6 CONCLUSION AND SCOPE OF FUTURE

6.1 Conclusion

The development of the arithmetic coding scheme was an important theoretical improvement over the traditional Huffman coding. However, as anecdotally demonstrated by the small number of sample compression factors above, the real-world benefit of the raw algorithm isn’t anything spectacular.

Nonetheless, but algorithms form important bases for more sophisticated work. In particular, the algorithms as presented are known as 0-order… they don’t base any of their symbol frequencies on symbol combinations. For most real-world data, the previous symbol can be a very good predictor of the next symbol. For instance, in C source code a right curly brace “}” is almost always followed by a carriage return.

There are n-order implementations of both algorithms that achieve much better compression by leveraging such statistics. However, such an adaptation is far from trivial and requires considerable programmatic book-keeping skills.

6.2 Scope for future work

There is an enormous amount of research in compression today. Most of it is directed toward lossy compression, which finds applications in video and sound compression technologies. However, lossless compression will always play a vital role, and indeed most lossy compression algorithms are actually lossless algorithms that just discard a few bits at some point.

An excellent follow-up paper for this one would be to investigate the more sophisticated versions of both Huffman and arithmetic coding algorithms. With dynamic frequency calculations and/or n-order prediction statistics, both algorithms achieve much better compression ratios and are better able to compete with today’s fancier tools.

Fractal image compression is still in its infancy and there are many aspects of this fast developing technology which are still partially unsolved, or untried. There are many opportunities to develop the ideas further and improve upon the methodology. Below are a few areas which could be explored.

- The geometry of the domain and range blocks need not be square, or even rectangle. This was just an imposition made to keep the problem tractable. Other geometries which might yield less blockiness, could be considered.
- The size of domain blocks, size of range blocks, size of the pool of range blocks and the number of allowable affine transformations was by no means optimized, and this could be investigated.
- A different distance metric between blocks could be used.
- Quadtrees could possibly be used with different sized blocks and multi-level image partitions.
- Work could be done to speed the compression stage up, e.g. possibly testing the range blocks in the locality of the current domain block first, or somehow eliminating a range block earlier in the process if it is unlikely to be a 'best-fit'.
- The pool of range blocks could be split into different sets; e.g. smooth blocks, edge blocks and midrange blocks. For a given domain block, the respective category is then searched for the best match.

7 References

[1] "Arithmetic Coding + Statistical Modeling = Data Compression" by Mark Nelson.
Dr Dobb's Journal, February, 1991.

[2] "A Mathematical Theory of Communication" by C.E. Shannon.
The Bell Systems Technical Journal, Vol 27, Jul/Oct 1948, pp. 379-423, 623-656

[3] "A Method for the Construction of Minimum-Redundancy Codes" by David Huffman.
Proceedings of the IRE, Vol. 40, No. 10, Oct 1952, pp. 1098-1101

[4] http://www.mathworld.com/

[5] http://datacompression.info/

[6] http://www.nist.gov/dads/

End Project Documentation

1 Configuration

a)Hardware: 128 RAM , 8GB Memory

b)Software: Window 2000/Xp, C/C++ , CMD

2 Enhancements

This project is for compression and decompression of text files but this project can also be enhanced for compression and decompression of files, folders, pictures (jpeg) and other text files. Depending on what we want to compress or decompress the coding can be changed accordingly. Thus we can compress and decompress loads of stuff. The compression rate and decompression rate can be enhanced to meet the users requirements.

3 Conclusion

After all the time and hard work put in by the Expendables, we have been successfully able to compress and decompress text files by almost 15-30% of the actual size. There’s scope for improvement by which we can modify the project program coding to compress and decompress files, folders, pictures but in order for that to happen it would require even more deep research into the field of science engg. and technology.