INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9200035

An error-free image compression algorithm using classifying-sequencing techniques

He, Duanfeng, Ph.D.

The University of Arizona, 1991

Copyright ©1991 by He, Duanfeng. All rights reserved.

U·M·I

**300 N. Zeeb Rd.
Ann Arbor, MI 48106**

**AN ERROR-FREE IMAGE COMPRESSION ALGORITHM
USING CLASSIFYING-SEQUENCING TECHNIQUES**

**by
Duanfeng He**

Copyright © Duanfeng He 1991

A Dissertation Submitted to the Faculty of the
OPTICAL SCIENCES CENTER
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 1

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

2

As members of the Final Examination Committee, we certify that we have read
the dissertation prepared by Duanfeng He

entitled An Error-Free Image Compression Algorithm Using
Classifying-Sequencing Techniques

and recommend that it be accepted as fulfilling the dissertation requirement
for the Degree of Doctor of Philosophy.

Eustace L Derenisak

July 12, 1991
Date

Michael D. Marcell

7/12/91
Date

R. Strublan

7/12/91
Date

Date

Date

Final approval and acceptance of this dissertation is contingent upon the
candidate's submission of the final copy of the dissertation to the Graduate
College.

I hereby certify that I have read this dissertation prepared under my
direction and recommend that it be accepted as fulfilling the dissertation
requirement.

Eustace L Derenisak

Dissertation Director

July 30, 1991
Date

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED:

Dedication

For Yunyun who joined my life in the year of this work

and

For my fellow Chinese people in search for freedom

Acknowledgement

I am grateful to my committee members for their help with this dissertation. I am especially indebted to Prof. Eustace L. Dereniak who provided unfailing support to me during the past four and a half years of his advisement. At the initial conception of this algorithm it was he who encouraged me to turn a pastime interest into a dissertation project. My sincere thanks go to Prof. Michael W. Marcellin whose careful reading and critical comments benefited me greatly in the revision process. I would also like to thank Prof. Robin N. Strickland for his useful comments.

My special appreciation goes to Prof. Murray Sargent III who answered my numerous questions regarding the use of PS word processor and provided timely upgrades which make possible the elegant graphs in this dissertation.

Many of my fellow students have offered generous support during this dissertation project. I would like to particularly thank Mark A. Voelker and Robert E. Uber who enthusiastically exchanged their ideas with me from the earliest stage. I would also like to extend my gratitude to Chihli Chuang, Doug Rautenkrenz, Mike Descour, Mark Shi Wang, Hong Tang, Thorsten Graeve, Mark Fink, Laurence Flath and Chong-Im Kim for their kindness and support.

Table of Contents

List of Illustrations	8
List of Tables	9
Abstract	10
1. Introduction	12
1.1 Definition of the Terms	14
1.2 Error-Free Gray-Level Image Compression	17
1.3 Brief Description of the Existing Algorithms	21
1.4 Brief Description of the Sample Files	25
2. Several Existing Algorithms	34
2.1 Huffman Code	34
2.2 <i>B</i> -Codes	36
2.3 <i>S</i> -Codes	38
2.4 Run Coding	40
2.4.1 Bit-Plane Run Coding	41
2.4.2 Gray Code	43
2.4.3 Alternative Run Coding	44
3. The Algorithm Using the Classifying-Sequencing Techniques	47
3.1 Overview	47
3.2 The Difference Classifying Process	49
3.3 Sequence Forming and Merging	52
3.4 Coding of the Sequences	55
3.4.1 Coding of Class 0 Sequences	55

Table of Contents (Continued)

3.4.2 Coding of Regular Class Sequences	56
3.4.3 Coding of Full-Length Class Sequences	57
3.5 Test Results on the Seven Sample Files	58
3.6 Discussions	59
4. Revisions to the Classifying-Sequencing Algorithm	62
4.1 The Floating-Buffer Technique	62
4.2 Choosing a Sequence Merging Strategy	64
4.3 Three and More Sequence Mergers	65
4.4 The Floating-Class Technique	68
4.4.1 The Hard Floating-Class Method	70
4.4.2 The Soft Floating-Class Method	71
4.5 The Simple Modulo Technique	73
5. Conclusions	75
Appendix A The Seven Sample Images	79
List of References	83

List of Illustrations

Figure

1-1	Separation of Source and Channel	21
1-2	Reciprocity Characteristic of the Conditional Probabilities	30
1-3	Shift-Insensitivity Characteristic of the Conditional	32
2-1	The Huffman Coding Algorithm	35
4-1	An Example of Effects of Different Orders for Sequence Mergers	64

List of Tables

1.1 Statistics on the Seven Sample files 27

1.2 Test of the Reciprocity Characteristic of the Conditional Probabilities 29

1.3 Shift-Insensitivity Characteristic of the Conditional Probabilities 31

2.1 Performance Evaluation -- Huffman Code 35

2.2 Performance Evaluation -- B-Codes 38

2.3 Performance Evaluation -- S-Codes 40

2.4 Performance Estimation -- Bit-Plane Run Coding 43

2.5 Performance Estimation -- Gray Code Run Coding 44

2.6 Performance Evaluation -- Alternative Run Coding 46

3.1 Classes of 8-Bit Images 50

3.2 Examples of Sequence Mergers 54

3.3 Performance Evaluation -- Classifying-Sequencing Method 58

4.1 Performance Evaluation
    -- The Classifying-Sequencing Method, with Different Implementations 66

4.2 Examples of Three-Sequence Mergers 67

4.3 Performance Evaluation -- The Hard Floating-Class Method 70

4.4 Performance Evaluation -- The Soft Floating-Class Method 72

4.5 Performance Evaluation -- The Simple Modulo Technique 74

5.1 Performance Comparisons 76

5.2 Comparisons Between Huffman Code and the Classifying-Sequencing Code 77

Abstract

Digital image compression is more and more in demand as our society becomes more information oriented, with more digital images being acquired, transmitted and stored everyday. Error-free, or non-destructive, image compression is required in applications where the final image is to be analyzed digitally by computers.

A new error-free digital image compression algorithm, the Classifying-Sequencing algorithm, is presented in this dissertation. Without the help of any statistics information of the images being processed, this algorithm achieves average bits-per-pixel close to the entropy of the neighboring pixel difference. In other words, the compression results are comparable to the best that a statistics code can achieve. Because this algorithm does not involve statistical modeling, generation of a code book, or long integer/floating point arithmetics, it is simpler and therefore faster than the standard statistics codes, such as Huffman Code or Arithmetic Code.

In this dissertation the new algorithm under discussion is tested using seven images, together with several known algorithms. Three lower-order entropies of the image files are also provided for comparisons. Presenting compression results from an isolated algorithm is not sufficiently objective for comparisons between algorithms, as potential discrepancies exist between not only different images but also same images when reproduced from prints. Comparing the results of different algorithms and with the entropy of the neighboring pixel differences on the same images is more objective. When the entropy of an image is high, the compression ratios of all algorithms are likely to be low; and vice versa.

Given that it is faster in decoding than in encoding images, the most prospective applications of the Classifying-Sequencing algorithm are in the fields of digital image

transmission, distribution and archiving, where the images are likely to be encoded once but decoded many times. It can be easily realized on simple processors, or completely in hardware, due to its simplicity.

Chapter 1

Introduction

Digital image compression is a very active field of research and application. There are two kinds of image compression algorithms: those that retain all the information in the image and those that do not. The majority of the compression algorithms developed do not keep all the information in an image. If the final image is to be inspected by human eyes, all the information contained in the image does not need to be retained. Pratt [1978] (p.589) states that “psychophysical experiments indicate that a human only comprehends visual information at a rate of about 50 bits per second. Also, information-theoretic studies show that most natural imagery is highly redundant. Nevertheless, conventional television coding systems present about 50 million bits per second.” Algorithms that do not retain all the information in an image include the Pulse Code Modulation (PCM) coding (Goodall [1951], Kretz [1975]), Deltamodulation (Schindler [1970]), Differential Pulse Code Modulation (DPCM) systems (O’neal [1966], Millard and Maunsell [1971], Bauch et al. [1974]), and all transform coding techniques (Clarke [1985], for example).

Increasingly, scientific and technical applications demand error-free (also known as non-destructive, reversible, noiseless, distortion-free, or lossless) image compression algorithms. Such applications include the storage, archiving, transmission and distribution of digital images in remote sensing, astronomy, radiology, and research image documentation. Grim [1980] pointed out that “For computer processing [of digital images] it is not known *a priori* what kinds of degradation would be permissible and therefore generalized compression cannot allow any degradation to occur.” Chung, Krasner and Mun [1990] stated that “liability issues, possible FDA guidelines, and acceptability to radiologists make adoption of irreversible techniques [in the field of medical image storage and archiving]

unlikely in the near future.”

Several error-free image compression algorithms are developed in the past. Section 1.3 gives a brief description of some of the known error-free gray-level image compression schemes. Among these several are given more detailed analysis and are their performances evaluated numerically in Chapter 2.

To evaluate an error-free image compression algorithm, it is not enough to only provide the pictures of the sample files, as so much detail is lost in the process of picture reproduction that the duplication may be meaningless. It is more informative when these algorithms are compared with one another, to a “standard” algorithm, or to any statistical values of the images used as samples. Many of the existing error-free image compression algorithms, however, have not been evaluated this way. Chapter 2 provides a comparison of many algorithms. The statistical data on the image files used in the comparison process are given in Section 1.4.

Chapter 3 introduces a new error-free image compression algorithm. Being a non-statistical algorithm, it has low demand on computation. The compression ratios it provides are comparable to the results of the Huffman Coding, the *de facto* “standard” compression algorithm in the field in the field, which is both reasonably fast and efficient in terms of compression ratios.

Several 8 bit-per-pixel image files are used throughout this dissertation as standard samples to evaluate the different compression algorithms. Therefore, for the sake of clarity and consistency, in this paper the algorithms are to be explained and illustrated in terms of 8 bit-per-pixel images wherever necessary, although not always explicitly acknowledged. These algorithms, however, can be applied to images of any word length.

1.1 Definition of the Terms

Different people may choose to use different terminology to mean the same things, and the same terminology to mean different things. There is not necessarily any system that is better than all others, but the possible confusion should be avoided. The field of error-free image compression especially lacks a standard set of terminology. This is exemplified by the fact that so many terms have been used to describe “error-free”. It is my intention to use the following terms consistently throughout this dissertation.

A *Message* is a piece of information as the input to a compression system. Naturally for error-free data compression systems it is also the information as the output from the decompression process. For image compression systems, depending on the coding technique used, a message word can be either the pixel element value, or the neighboring pixel difference value. Sometimes an algorithm groups several message words together to make a combined message and compresses this combined message instead of the individual message words.

A *Code* is a piece of information as the output of the data compression system. Code words in a coding system can be of either variable length or fixed length. Some algorithms use one code word to code each message word, some others use a variable number of code words to code different messages.

A *Code Series* is the several code words in sequence used by an algorithm to code a message. A message may be a simple message word, or a complex message composed of several consecutive message words.

The *instantaneousness* of a compression algorithm is one of the qualitative measures on its speed. It is specific to either the encoder or the decoder. The encoding process is instantaneous if it can code its incoming message word with only the knowledge of the current and past messages. The decoding process is instantaneous if the decoder can process

the current code word without the information of what follows in the input code stream. For some algorithms the encoder is instantaneous; for some others it is semi-instantaneous, in which case the encoder has to work on several message words at a time, but do not need to scan through the whole image; for the rest it is non-instantaneous, and the encoder has to scan over the whole message set (a complete image file, for example) before it can code the first message word. Most image compression algorithms are instantaneous at the decoder.

The *entropy* of a message word is the amount of information carried in the message word. This amount depends on the probability distribution of the ensemble from which the message is sampled. If a message word is less probable to appear, its occurrence is more informative, and vice versa. Mathematically the entropy of a message word is given by (Shannon [1948]):

$$H_w = - \sum p_i \cdot \log p_i, \quad (1.1)$$

where i takes the value of any probable message word and p_i is the probability that a message word is i . When the logarithm base is 2 the calculated entropy result is in units of bits. For an image, the messages to be coded can be the pixel values themselves, or more often, the neighboring pixel differences. The probability of a message word is calculated from statistics performed over the actual image.

The entropy of an image is the amount of information carried by all the elements that make up this image. Mathematically, if we assume that this image can be represented by N message words in a series:

$$d_1, d_2, \dots, d_N, \quad (1.2)$$

then the entropy of the image is:

$$\begin{aligned}
H = & - \sum p(d_1) \cdot \log p(d_1) \\
& - \sum p(d_1, d_2) \cdot \log p(d_2|d_1) \\
& - \sum p(d_1, d_2, d_3) \cdot \log p(d_3|d_1, d_2) \\
& \cdot \\
& \cdot \\
& \cdot \\
& - \sum p(d_1, d_2, \dots, d_{N-1}, d_N) \cdot \log p(d_N|d_1, d_2, \dots, d_{N-1})
\end{aligned} \tag{1.3}$$

where $p(d_1)$, $p(d_1, d_2)$, $p(d_1, d_2, d_3)$, ... $p(d_1, d_2, \dots, d_{N-1}, d_N)$ are joint probabilities, and $p(d_2|d_1)$, $p(d_3|d_1, d_2)$, ... $p(d_N|d_1, d_2, \dots, d_{N-1})$ are conditional probabilities. The summations are taken over all possible combinations of values. But the entropy as defined in Equation (1.3) is difficult to calculate, because long range correlations do exist among image pixels, and the statistics for the conditional probabilities are therefore difficult to perform. Most often it is assumed, without proper justification, that every message word in an image is independent, and thus the conditional probabilities in Equation (1.3) are the same as the non-conditional probabilities as represented in the first term of the equation. Under this assumption, every message word carries the same amount of information, which in turn is given in Equation (1.1).

In discussions to follow, I often regard the pixel values in an image as a one-dimensional string of N values, as shown in Series (1.2), instead of viewing the image pixels as in a two-dimensional array. I also refer to neighboring pixel differences frequently.

Because of the correlations in a natural data source such as an image, the entropy of a difference word is expected to be smaller than that of a pixel word (see data in Section 1.4). For simplicity in the comparisons I use the one-dimensional string model in Series (1.2) for the image words, and the "neighboring pixel differences" are defined as

$$\delta_k = d_{k+1} - d_k \quad (k = 1 \dots N-1) \tag{1.4}$$

Similarly, the difference between the neighboring differences, or the second order differences, is

$$\delta\delta_k = \delta_{k+1} - \delta_k \quad (k = 1 \dots N-2) \quad (1.5)$$

1.2 Error-Free Gray-Level Image Compression

The concept error-free image compression in this context means a system composed of an encoder and a decoder. The encoder takes a digital image and record it in less memory than the original form, or “compresses” the image. The decoder on the other hand takes this compressed image and produces an exact replica of the original image. The replication should be exact not only to human visual inspection, but also to a digital computer, which compares the two images bit by bit. Gray-level images are those which have more than two shades. Usually a gray-level image contains 64 different shades or more.

In the input to a data compression system, the messages usually come in fixed word lengths (e.g., a character, such as an alphabetical letter, a numerical digit, or a punctuation mark, is represented by an eight bit word in many computers). For its output, however, a code system can use fixed or variable code word lengths. Depending on the nature of the individual algorithm, it may choose to combine a variable number of message words to treat at a time, thus producing a variable “word” length message source from a fixed word length message source. Images are usually stored such that each pixel is recorded by one computer word.

Source coding can be differentiated according to the forms of the message set that can be coded and the code series used to code each message (Tjalkens [1987]). The output of many data compression systems are codes of variable word length. In other words, fixed-length messages are coded into variable-length codes. These algorithms can be referred to as FV algorithms. Similarly there are VF schemes, in which variable-length messages are

transformed into fixed-length codes, and VV schemes, in which variable-length messages are translated into variable-length codes. FF schemes are not very practical and are seldom discussed.

The most famous unequal word length message system is the Morse code. A Morse code word contains a variable number of dots and/or dashes, with short spaces separating these dots and dashes. A longer space is used to separate the code words from each other. Over all, we can regard the short space as the separator between all the symbols. Therefore the Morse code system is a ternary system, with three individual symbols: the dot, the dash, and the long space. With a binary system, which is universally used in the computer industry, it is not cost-effective to reserve one of the symbols for the code word separator. To design a variable-word-length code system that does not have a word separator, a message dictionary (or code book) is generated before the messages are coded. In this dictionary every every possible message word is assigned to a distinctive code word. To facilitate the decoding process, usually the dictionary does not contain code words which are the prefix of other code words.

To demonstrate the concept of compact code, or a variable-word-length code system that does not have a word separator, we consider an example in written English. This would give us a clearer idea as how it is possible to construct such a system, as well as its limitations. Suppose we get an English dictionary, and based on it we want to construct a code series without word separators, which naturally is the “space” used in a regular passage like this sentence. We first need to trim the dictionary, crossing out words such as “there” and “head”, because they each contains a prefix that is by itself an entry in the dictionary (“the” and “he”, respectively). These words should not be used, because the decoder would have difficulty in finding where the word ends. After finishing this task, we have a limited vocabulary but can still make complete code series, such as this one:

“themoStfamousunequalwordlengthmessagesystemistheMorsecode”.

Apparently, this series of text can be uniquely decoded into the original code words. But parsing such a code series into separate code words can be time consuming. Because the decoder does not know the word length of the next code word, usually one test is required for each bit of code series to decide whether it is the end of the code word.

Languages and their derivatives (shorthands, computer languages, etc.) are human designed semiotic systems and are thus considered artificial data sources. The continuous measurements of a physical quantity (such as a sequence of temperature readings of a certain place taken at a constant time interval), on the other hand, constitute a natural data source. An artificial data source can be, and usually is, very different from a natural data source. While the conditional probabilities of a pair of messages in a natural data stream is usually the close to that of the reciprocal conditions, this is not true for artificial data sources (e.g., in a stream of English language text it is expected that the probability of t followed by h is higher than h followed by t). Also, while in a natural data stream the conditional probabilities are shift-insensitive, in an artificial data stream the opposite is usually held (e.g., the probability of t followed by h is not the same as u followed by g).

An image data stream has characteristics very similar to those of a natural data stream. The correlations between two neighboring data words are reciprocal and shift-insensitive. Usually higher order correlations exist as well as the correlation between the neighboring pixels. What is specific about an image data stream is that it is correlated in two dimensions, if the data are arranged in a pre-defined fashion into a two-dimensional matrix. When a stream of image data is not restored into a two-dimensional matrix, it behaves much like an average natural data stream with occasional sudden jumps corresponding to the change of scan lines.

To evaluate the amount of information in an image file I frequently refer to

neighboring pixel differences. For easy evaluation of the algorithms, I choose to regard image files as organized in a one-dimensional stream, and the difference values are actually obtained from the adjacent elements in this one-dimensional data stream, rather than in the image field. In other words, the line-breaks between adjacent scan lines are ignored.

The algorithm introduced here, and the other algorithms I evaluate in this paper for comparative purposes, are error-free gray-level image compression algorithms. Specifically designed, these compression algorithms are more suitable for images, or other natural data streams, since they do take advantages of the special properties of the natural data source that may not be available in artificial sources. But because they are error-free, they can be applied to general data compression applications (Contour Encoding, which uses the two-dimensional characteristics of the images, is not evaluated in this paper). However, if some other data is processed by these systems (e.g. an executable computer program), the decompression process still returns this data in its original form, but the “compressed” form is not necessarily smaller than the original uncompressed form.

Noise impact on the compression algorithm is one of the directions that merits research attention. Here I am referring to the noise that is mixed into the image before the image is encoded by the image compression program (or hardware). Lo, Krasner and Mun [1990] gave a good example as how this kind of evaluation could be executed. However they only dealt with the effects of additive random noise, while neglecting the possibility of a multiplicative noise. Multiplicative noise can be generated by an input device such as a CCD imaging array in a video camera. On the other hand, the general observation that an image file should not retain more bits than that granted by the signal-to-noise ratio of the imaging device, or, depending on the application, the digitizing device, is understandably universal.

Generally speaking, it may not be of the same degree of concern to evaluate the noise

impact of the communication channel. It is true that the image compression algorithms remove some redundancy in an image and therefore subject it to more severe noise impacts during the communication channel transmission process. But this does not justify that the redundancy be retained. The separation of source encoder and channel encoder has been discussed early on with the information theory (Reno [1961]). In short, the source encoder should take out the random redundancy in the source stream, while a controlled process at the channel encoder adds redundancy such that the minimum additional bits are used to facilitate error detection/correction at the channel decoder (see Figure 1-1). Then the source decoder generates a replica of the original data stream from the output of the channel decoder. In the fields of digital image transmission, distribution and archiving, where error-free image compression algorithms are most likely to be used, it is common practice that the hardware or hardware/software combination in the transmission or storage device have built-in channel encoder-decoders.

Figure 1-1 Separation of Source and Channel

1.3 Brief Description of the Existing Algorithms

In this section an overview is given to several existing algorithms used in error-free

image compression. Chapter 2 gives detailed analysis of several important ones among these, together with their performance evaluations.

The first coding algorithm devised with the information theory is the Shannon-Fano Code (Shannon [1948] and Fano [1949], or Pratt [1978]). The average code length required in Shannon-Fano Code is no more than 1 bit above the average amount of information (entropy), and at special instances can be the same as the message entropy. The coding method is easy to understand and easy to implement. For coding with binary code systems, the algorithm first separates the possible message words into two groups with equal, or as close as possible, total probabilities. Every message in the first group is then given the code prefix of 0, and every message in the second group is given the code prefix of 1. The same is then done in each of the subgroups, recursively, until all message words are given distinctive code words. This coding system is an FV algorithm and is instantaneous at the decoder. Because the best achievable compression ratios can only be derived from the exact probability distribution of the message ensemble to be transmitted, a statistical scan over the message ensemble is required before the encoder can code the first message word, which means this algorithm is not instantaneous at the encoder. Since the development of the Huffman Code (see description below and in Section 2.1) the Shannon-Fano Code becomes obsolete. This is because the Shannon-Fano Code has similar requirements and complexity as the Huffman Code, but yields less (or, at rare occasions, equal) compression efficiency.

The most important, and most widely used, FV algorithm in error-free image compression is the Huffman Code (Huffman [1952]) applied on adjacent element differences. As mentioned earlier, because of the correlations between neighboring pixels, the adjacent element differences for an image file have a smaller entropy than that of the original image data values. When each of these difference values are treated as independent from the others, the Huffman Code produces the smallest average code length for FV

schemes, provided that the probabilities of these difference values appearing in the particular image file is known before the code book is generated. These probabilities are generated by a pre-coding scan of the image, and thus the Huffman Code is not instantaneous at the encoder. To the decoder, the next code word has an unknown length. The process to generate the Huffman code book for a given set of probabilities is illustrated in Section 2.1, together with the performance evaluation the Huffman Code.

The Lempel-Ziv (or LZ) algorithm (Ziv and Lempel [1977]), or its revision, the Lempel-Ziv-Welch (or LZW) algorithm (Welch [1984]), is also an algorithm that generates code books. Unlike the Shannon-Fano Code or the Huffman Code, this algorithm does not require the complete statistical knowledge of the message ensemble being coded. In contrast to the other two mentioned above, both the encoder and decoder generate the code book “on-the-fly”. The algorithm records the history of the source probabilities, and assumes that the patterns found in the history of the source are highly likely to be repeated. Since a pattern in the source history is composed of a variable number of message words, this algorithm belongs to the VF or VV category, depending on the implementation. Both the encoder and decoder being instantaneous, this algorithm finds most application in file compression programs (Vaughan-Nichols [1990]). Intended for text compression, where the probabilities of the patterns are neither reciprocal nor shift-insensitive, the application of the Lempel-Ziv algorithm in image compression is not very successful (Lo, Krasner and Mun [1990]).

The *B*-Codes and *S*-Codes (Gonzalez and Wintz [1977]) are two families of FV algorithms with an interesting property. Under these algorithms the codes generated for a message are composed of one or more code words, while every code word has the same length. The design has the advantage over the Huffman code in that the decoder does not have to guess what the length of the next code word is, and therefore can be more time-

efficient. These algorithms can be instantaneous to both the encoder and decoder (see Sections 2.3 and 2.4 for more detailed discussions).

Papers discussing Run-Coding (Laemmel [1951]) comprises the majority of the papers on error-free image compressions. Depending on the implementation, Run-Coding can be either a VF (Pratt [1978]) or VV algorithm (Golomb [1966]). Run-Coding is based on the concept that if it is likely for many consecutive pixels to have the same value, then their value and the number of the consecutive pixels having this value can be coded. This algorithm can, of course, code the neighboring pixel differences, but the probability of adjacent difference values being equal is not very much higher, and having to use one more bit per pixel to record the neighboring pixel differences usually makes it impractical to code the differences. Most suitable for binary images, the Run-Coding encoder can easily produce “compressed” output files that are larger than the uncompressed original image files when dealing with gray-level images (Grim [1980]). To maximize the compression ratio, a statistics over the image is necessary, as for the Huffman Coding. Three revised Run-Coding implementations suitable for gray-level images are discussed and evaluated in Section 2.4.

The Contour Encoding (Gonzalez and Wintz [1977]) is a two-dimensional variation of Run-Coding. It traces out the contours in an image and the complete image is reconstructed with these contours. It achieves relative success on images with little high frequency components and few quantization levels. With the images that cannot be coded efficiently with Run-Coding (for example, see the performance of Run-Coding on our sample files, as shown in Section 2.4), the Contour Encoding method cannot succeed either. This algorithm is not evaluated in Chapter 2.

The Arithmetic Coding algorithm is first introduced by Elias (see Abramson [1963], pp. 61-62). This algorithm uniquely uses a single real number between 0 and 1 to code any

message string. If the message string is longer, more bits will be used to identify the real number. The first practical implementations were introduced by Rissanen [1976] and Pasco [1976], independently, and then further developed by Rissanen [1979]. A detailed implementation of Arithmetic Coding with progressive operation using integer arithmetic is given by Witten *et al.*[1987]. This algorithm resolves the coding inefficiency of FV coding schemes (such as Huffman Code), where the code of each message word has to be separated at bit boundaries, and thus achieves compression efficiencies arbitrarily close to the limit given by the appropriate entropies. The closeness of the compressed average bits-per-word to the entropy value is only limited by the word length of the integer type of the computer being used. The evaluation of Arithmetic Coding on the neighboring pixel difference coding of images is trivial, because the compressed bits-per-word should be exactly as given by the H_δ values in Table 1.1, if we use a long enough integer type (such as the 32 bit long integer available on personal computers) for the calculations. The Arithmetic Coding is best for dynamic modeling, which solves the problems of two-scan processing and having to transmit the code book. Although dynamic modeling does not necessarily sacrifice compression ratios (Cleary and Witten [1984]), it does cost more computing. Also, Arithmetic Coding has higher computational demand on the decoder than the encoder. Many recent development of Arithmetic Coding eliminates the multiplication processes, which increases both the encoding and decoding speed (Pennebaker *et al.* [1988] shows one revision dealing with binary encoding, while Chevion *et al.* [1991] discusses a more general scheme), but the burden of doing dynamic statistics still remains.

1.4 Brief Description of the Sample Files

To evaluate the algorithms in a way as meaningful as possible, seven common sample files are used throughout this dissertation. These seven images are images of 8 bit word

length. The image pixel values are unsigned, with 0 representing black and 255 representing white. The pictures of the seven images are provided in Appendix A.

Of these seven sample files, Gold is an STM (Scanning Tunneling Microscope) image of the surface of a gold film, Mark and Steve are two head portraits, Demo1 is a partial image of the Statue of Liberty, Demo2 is a satellite picture of earth, Demo3 is a still shot of an apple and an orange, and Demo4 is the image of part of a dollar bill.

The images are stored in Data Translation image format. In this format, there is a fixed portion of 512 bytes at the beginning of every image file, recording the image parameters, dates of the original and revision, comments, among other things. File Gold is a blown up and smoothed version of a smaller original, and has an image size of 243×257 pixels. Files Mark and Steve are of 512×480 pixels. Files Demo1 through Demo4 are sample files prepared by Data Translation. These files contain a visible portion of 512×480 pixels, but are padded with identical elements at the end to fill up a 512×512 format.

It is a common practice for an image compression article to print the pictures used in the article. However, the algorithms discussed here are error-free ones, and that neither the printing or the visual inspecting process can retain the details of an image nearly error-free. Therefore, in addition to the pictures of the sample images provided in Appendix A, several of the lower order entropies of the files are provided here. Together with the evaluations using Huffman code (as well as several other error-free codes), the information provided here should be more descriptive of the images than prints of these images could be. I also suggest that any evaluation on an error-free image compression algorithm should at least provide the lowest two orders of entropies of the images, and the test results of the Huffman code, along with the usual practice of providing the test result of the algorithm being discussed.

The statistics provided here are done by treating each image file as a one-dimensional

data stream. That is, each and every byte in the files is counted into the statistics, whether or not it belongs to the actual image area. Accordingly, every algorithm evaluated in Chapter 2 is also implemented in this fashion. The advantages are that the compression process becomes simpler, as the encoder and decoder do not have to know the image format, and that the compression and decompression processes can be faster. In fact, the same encoder and decoder are used to process SPOT satellite images, which are stored in 8-bit-per-word binary form without headers.

In Table 1.1 N is the size of the files in words, including the 512 byte header. H_d is the data entropy, H_δ is the entropy of the neighboring pixel differences, and $H_{\delta\delta}$ is the entropy of the differences of the neighboring pixel differences, or the second order neighboring pixel differences. The entropies are in units of bits. The three entropies are all calculated from Equation (1.1), but with different kind of information as the messages. For H_d the message words are the data values as given in Series (1.2). For H_δ they are the neighboring pixel differences as given by Equation (1.4). And for $H_{\delta\delta}$ they are the second order neighboring pixel differences given by Equation (1.5).

Table 1.1 Statistics on the Seven Sample files

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
N	62963	246272	246272	262656	262656	262656	262656
H_d	7.96	7.63	6.41	6.56	6.86	7.12	6.80
H_δ	5.51	3.86	4.54	4.74	4.64	3.61	5.08
$H_{\delta\delta}$	4.04	3.88	4.89	4.87	4.69	3.95	5.21

The H_d values in Table 1.1 are close to but below 8 bits. For 8-bit-per-pixel images, they cannot exceed 8 bits. If the image pixels are uniformly distributed among the 256 possible values, then H_d is 8 bits. If half of the possible values are not used, the H_d cannot be larger than 7 bits.

The H_δ values are smaller than the respective H_d values. This is because of the correlation between neighboring pixels. If the image is completely random (such as a speckle image), then the H_δ value should also be 8 bits, just as the H_d value (assuming the image is recorded in 8 bit-per-word format).

On average, the $H_{\delta\delta}$ values are smaller than the H_d values, but larger than the H_δ values. This is because that there is the longer range correlation between three neighboring pixels, but it is not as strong as the two-neighbor correlation.

Earlier in Section 1.2 it is mentioned that for a natural data source the conditional probabilities have the reciprocity and shift-insensitivity characteristics. Here I examine these characteristics, using the seven sample image files.

In Table 1.2 results are given to verify the reciprocity characteristic. This characteristic implies that for any pixel value having a given value m , the statistics of the pixel just before it should be the same as that of the pixel right after it. Arbitrarily we choose $m=100$. To collect a large enough sample base for Table 1.2 statistics done on all the seven sample files are summed together.

The conditional occurrence number N_j^i used in Table 1.2 is defined as:

$$N_j^i = N(d_k=i, d_{k-1}=j) \quad (1.6)$$

and the occurrence difference ΔN is calculated as:

$$\Delta N = N_{100}^i - N_i^{100} \quad (1.7)$$

The reciprocity characteristic can be expressed as:

Table 1.2 Test of the Reciprocity Characteristic of the Conditional Probabilities

i	N_{100}^i	N_i^{100}	ΔN
90	116	84	32
91	102	66	36
92	155	86	69
93	114	123	-9
94	271	219	52
95	143	137	6
96	323	334	-11
97	300	320	-20
98	575	570	5
99	501	473	28
100	571	571	0
101	503	494	9
102	512	533	-21
103	255	272	-17
104	328	318	10
105	186	192	-6
106	189	167	22
107	98	119	-21
108	132	135	-3
109	74	81	-7
110	71	124	-53

$$N_j^i \simeq N_i^j \quad (1.8)$$

Because the statistics are done on only seven different images, and that in each image data value occurrences are correlated, the statistics as given in Table 1.2 are not necessarily representative to all images. Still, we can see that the statistics of N_{100}^i and N_i^{100} as listed in Table 1.2 closely resembles each other. Figure 1-2 gives a graphical view of the reciprocity characteristic.

Table 1.3 provides the statistical data collected over all seven sample images for verifying the shift-insensitivity characteristic of the conditional probabilities. The values of

Figure 1-2 Reciprocity Characteristic of the Conditional Probabilities

j being 100 and 200 for the center two columns are chosen rather arbitrarily. As the range for the δ_k values listed in Table 1.3 is from -10 to +10, the j values of 10 and 245 are chosen so that their distance to the nearer data range is 10. This makes it possible for every bin in the statistics to collect a non-zero count.

In Table 1.3 the summations are done on all the possible conditions, rather than only on those occurrences that falls in the selected δ_k values that appear in the table:

$$\text{sum}\{ N_m^i \} = \sum_{i=0}^{255} N_m^i = N(d_k=m) \quad (1.9)$$

Figure 1-3 shows graphically the shift-insensitivity characteristic of the conditional

Table 1.3 Shift-Insensitivity Characteristic of the Conditional Probabilities

δ_k	N_{10}^i	N_{100}^i	N_{200}^i	N_{245}^i
-10	0	116	92	31
-9	0	102	61	40
-8	1	155	104	41
-7	0	114	88	47
-6	0	271	135	38
-5	1	143	91	77
-4	7	323	206	71
-3	17	300	295	208
-2	12	575	643	283
-1	43	501	561	224
0	70	571	598	152
1	93	503	540	137
2	131	512	512	113
3	70	255	395	152
4	13	328	231	129
5	8	186	97	47
6	17	189	127	49
7	3	98	117	34
8	2	132	112	32
9	3	74	77	44
10	0	71	85	124
<i>sum</i>	502	6737	6043	2305

probabilities. The conditional probabilities are given by:

$$p_m^i = \frac{N_m^i}{\text{sum}\{N_m^i\}} \quad (1.10)$$

The shift-insensitivity characteristic proclaims that for any integer b the following condition holds:

$$p_m^i \simeq p_{m+b}^{i+b} \quad (1.11)$$

provided that the four related occurrences are all possible occurrences.

When discussing about the reciprocity characteristic I explained that the seven image

Figure 1-3 Shift-Insensitivity Characteristic of the Conditional Probabilities

files are not quite enough to make a good statistic to represent all possible images, which holds true for the tests for the shift-insensitivity characteristic as well. If more images are used in the statistics, the resultant curves resemble each other better, and they all tend to a bell-shaped curve peaked at $\delta_k=0$. In Figure 1-3 the line shape of p_{10}^i is more different than among the others, partly because the $\text{sum}\{ p_{10}^i \}$ is comparably even smaller than the sum values for the other curves, therefore the statistical result is not as representative. Note that for each value of m there are 256 values that δ_k can take. Considering this large range, the concentration near $\delta_k=0$ is fairly significant in Figure 1-3.

As the reciprocity characteristic, the shift-insensitivity characteristic is an approximation. Every image file has its specific characteristics in the conditional probabilities because of the repetition between different linescans. Therefore, the statistics conducted on the seven of sample image files do not give us a conclusive line shape to which the conditional probability distribution function is to converge to, but the existence of these relations between the conditional probabilities are visible in the figures presented here. A good algorithm that deals with images could utilize these properties, but should not expect that the probabilities of the reciprocal or shifted conditions to be exactly the same.

Chapter 2

Several Existing Algorithms

2.1 Huffman Code

The Huffman Minimum-Redundancy Code (Huffman, [1952]) is a general information coding system that can be applied to digital image compression. Usually referred to as the Huffman Code, it uses binary digits for coding a message, but can be expanded to use non-binary codes as well. When transmitting several independent messages from a message ensemble with known probabilities, the code thus generated is of the shortest possible average length. Huffman Code produces average bits-per-message-word values which approach from above the entropies of the incoming messages, and at some special cases can be the same as these entropies.

As pointed out by others (see Pratt [1978], for example), it is not practical to code a whole image file or even just a scanline of an image as a complete message, because first, the information contained in them is so huge that the code book itself becomes too long; and second, it is not practical to decide which image is more probable to appear than others. Coding each individual pixel element in an image is practical, but the yield may not be good, as the pixel value entropies are often close to the bits-per-pixel the image is originally recorded in. It is also seldom used because one can code the neighboring pixel differences just as easily, which requires a code book that is only twice as long, but the compression ratios achievable is higher. The entropy values given in Table 1.1 illustrates this point.

In the *C* language syntax, the Huffman Code generation procedure can be written in a recursive fashion as in Figure 2-1.

Table 2.1 gives the results of the Huffman Coding on our seven sample data files. The

Figure 2-1 The Huffman Coding Algorithm

```

HuffmanCode ( ensemble of  $N$  messages) {
  if ( $N==2$ ) {
    first message is assigned the code 0;
    second message is assigned the code 1;
    return;
  }
  find the two messages with the least probabilities;
  sum the probabilities of these two messages;
  form a new ensemble, with this summed up message and all others;
  HuffmanCode ( new ensemble of  $N-1$  messages);
  SumCode=the code for the summed up message;
  first of the original messages is assigned the SumCode followed by code 0;
  second of the original messages is assigned the SumCode followed by code 1;
}

```

coding is done on the neighboring pixel differences. In this table *Avg* is the average code word length in bits. The code book is generated on the statistics of each individual files. In practice the code book thus generated has to be transmitted so that the messages can be deciphered by the decoder. Alternatively, the histogram of the image can be transmitted, from which the decoder can regenerate the code book. The overhead to transmit the code book or the histogram of the file is not included in the *Avg* numbers in Table 2.1.

Table 2.1 Performance Evaluation -- Huffman Code

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
<i>Avg</i>	5.55	3.88	4.57	4.79	4.67	3.64	5.11

It can be seen that the *Avg* values given in Table 2.1 are close to and always larger than

the H_6 values in Table 1.1. If a compression algorithm codes each message separately in an integer number of bits, its performance is to be equal to or inferior than that of the Huffman Code. The respective compression ratios are calculated by dividing the Avg values by 8, which is the bit-per-word value for the original message data.

2.2 B-Codes

The B -Codes (Gonzales and Wintz [1977]) use multiple code words to encode each single message word. Each code word is of a fixed length, which facilitates the decoding process. B -Codes belong to the FV category.

The construction of the B -Codes is as follows. First, each individual messages are sorted according to their probabilities. The messages are then given code words in the descending order of their probabilities. Those messages with highest probabilities are given single code words, while those with smaller probabilities are coded with multiple code words.

For B_1 -Code, each code word consists of two bits. Therefore, group 1 uses one code word and contains two messages, group 2 uses two code words and contains four messages, up to group n which uses n code words to describe 2^n different messages. The total code length for a message in group n is $2n$ bits.

For B_2 -Code, each code word consists of three bits. Therefore, group 1 uses one code word and contains four messages, group 2 uses two code words and contains sixteen messages, up to group n which uses n code words to describe 4^n different messages. The total code length for a message in group n is $3n$ bits. Similarly, B_3 -Code uses four bits for each code word, and in which group n contains 8^n messages each coded with $4n$ bits.

Each code word is composed of two parts: a continuation bit and the actual code. In the original design, the continuation bit flips from message to message, to indicate which of

the several code words in a code series belong to the same message word. But this makes the process non-instantaneous: the decoder has to read the next code word in order to find whether the current message has completed.

Gonzales and Wintz [1977] state that the *B*-Codes are not instantaneous to the decoder, which hinders performance. A simple modification, however, can make the *B*-Codes instantaneous. The original algorithm separates code words for different messages by flipping the continuation bit from message to message. This can be modified by using 0 (chosen arbitrarily as an example) as the continuation bit for the end words of every message, and 1 (following the same example) for all the other words. For example, if a message is coded with four code words, the continuation bits of the first three code words are 1, and that of the last word is 0.

For comparison the *B*-Code algorithms are applied to the neighboring pixel differences, and the results on the sample image files are given in Table 2.2. To make the comparison relate more closely to the Classifying-Sequencing Method, the messages are not sorted according to the probabilities of their appearance in an image. Rather, an index is generated according to the neighboring pixel difference value:

$$i = \begin{cases} 2\delta & \delta \geq 0 \\ -2\delta-1 & \delta < 0 \end{cases} \quad (2.1)$$

It is assumed that values with lower indices are more probable to appear than those with higher indices. The results in compression ratios are then not the optimum values. But in return, we get the benefits of a non-statistical method: no statistics need to be performed on the image before it is coded, and no code table has to be transmitted.

Table 2.2 gives the performance of the *B*-Codes on our sample files. These again are

Table 2.2 Performance Evaluation -- *B*-Codes

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
B_1	6.40	4.25	5.32	5.02	5.04	3.91	5.58
B_2	6.28	4.54	5.45	5.20	5.21	4.32	5.62
B_3	6.54	4.70	6.10	5.46	5.52	4.60	5.93

average bits-per-word. On the average, the B_1 -Code seems to perform better than the B_2 - and B_3 -Codes. *B*-Codes of longer code word lengths can also be constructed. But the average code bits per message word cannot be shorter than the basic code word, of course. For example, the B_3 -Code has code words of four bits, and therefore the average bits per message word is to be larger than 4. The performance of the *B*-Codes are not as good as that of the Huffman Code (shown in Table 2.1), but *B*-Codes have the advantage of fixed code word length.

2.3 *S*-Codes

As the *B*-Codes, the *S*-Codes (or Shift-Codes) are designed for a set of messages with descending probabilities. It also uses more code words for those messages with small probabilities. The *S*-Codes, unlike the *B*-Codes, can be of unequal code word length, as illustrated in Gonzalez and Wintz [1977]. Like the *B*-Codes and the Huffman Code, the *S*-Codes encode each message word independently, and therefore the compression ratios it can achieve is bounded from above by the compression ratios of the Huffman Code. Since the only advantage of this coding scheme over the Huffman coding method is that of the fixed code length, the variable code word length alternative (as given in Gonzalez and Wintz [1977]) loses all its advantage, and is not discussed here.

The S_4 -Code contains code length of 4 bits each. In the original design, for coding neighboring pixel differences, the first group of 14 codes are to designate differences nearest to 0, namely from -7 to +6. The other two possible codes are used as *shift* words (hence the name of the algorithm). When the difference value is out of the range of the first group, a shift word is transmitted, indicating the actual difference value is that given by the code word to follow plus a bias. The bias is +14 if the up-shift word is transmitted or -14 if the down-shift word is transmitted. Multiple shift words are used in series if the difference value is still farther from 0.

Similar to the S_4 -Code described above, S_3 -Code, or even S_2 -Code, can be generated. In an S_3 -Code system for coding the neighboring pixel differences, each code word is 3 bits long. In an S_2 -Code system, the code words are 2 bits long.

The S -Codes can be revised easily to be more efficient. Since it is not possible to have an up-shift word followed by a down-shift word, or vice versa, we can revise our coding scheme so that each shifted group contains one more element. For example, in S_4 -Code, which the bias-free group can contain 14 difference values, the shifted groups can each contain 15 of them, which allows the average bits-per-word used to drop.

Table 2.3 gives the test results of the S_2 -, S_3 - and S_4 -Codes, using the revised scheme as that described in the previous paragraph. The test results are given in bits per word. As in the treatment for the B -Codes, the indexing method as given in Equation (2.1) is used for the generation of the S -Codes.

As shown in Table 2.3, S_4 -Code performs better on our sample files than the S_2 -Code and S_3 -Code on average. But the performance of S_4 -Code is not as good as that of the B_1 -Code, which is shown in Table 2.2. None of the S -Codes is as efficient as the Huffman Code (see Table 2.1), which is expected.

Table 2.3 Performance Evaluation -- S -Codes

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
S_2	7.98	4.61	6.31	6.53	5.51	3.87	6.77
S_3	6.77	4.32	5.74	5.70	5.11	3.89	5.95
S_4	6.17	4.52	5.61	5.39	4.92	4.23	5.51

S_2 -Code shows very poor performance in practice. This is because that the first group will still have to reserve two words for the shift words, and will only have two elements to be used to code messages. The inefficiency thus brought about is not tolerable. On the other hand, we can see that S_5 -Code is not desirable, at least for our sample files, since the basic code word is five bits long, which is more than the average code word length of many files under S_4 - or S_3 -Code.

2.4 Run Coding

Run coding is designed for images with few gray levels and where pixels of the same gray levels bunch together. Images satisfying this feature usually have too few gray-levels, and show visible contours. A run is a continuous series of pixels having the same values. In Run Coding if a pixel has a different value from both the previous pixel and the next pixel it is called a run (of unity length) by itself. Run coding is mostly used in facsimile transmissions where only two gray levels, namely black and white, are used.

To realize Run Coding, it is not practical to let runs with infinite length to be coded in one message. Practical coding is achieved by artificially breaking up a run, or creating *pseudo* runs. A pseudo run can be given a code as if it is a natural run of 0 length, signifying that the previous run is continued in the next code word. But the use of pseudo

runs inevitably decreases the overall performance, in terms of compression ratio, of Run Coding: a long run may have to be coded in more than one code words, and a special word has to be reserved for the 0 length runs of the opposite value from the current pseudo run.

There are a great number of implementations of Run Coding, with each implementation being different but not necessarily superior or inferior to all others in all aspects. Generally, for creating fixed word length codes, the optimum word length is dependent on the probability of a run occurring. When variable word length codes are used, the system becomes even more complex, and the design of the code book depends on the probability distribution of all the possible run lengths. The majority of the Run Coding implementations are designed for binary images. Here I describe and evaluate three implementations that specifically deal with gray-level images.

2.4.1 Bit-Plane Run Coding

When dealing with gray-level images, the complexity of the image can easily make the strict Run Coding to expand the image, rather than compressing it (Grim [1980]). The most often used technique is to split the image into bit-planes and each plane is dealt with individually as if it is an independent black-and-white (or binary) image (Schwartz and Barker [1966]).

If the source of a binary image is completely random, and that unity length runs are included in the calculations, then the probability for a run to have a length n is

$$p(n) = \frac{1}{2^n}, \quad (2.2)$$

and the average run-length is:

$$\bar{\mathcal{L}} = \sum_{n=1}^{\infty} n \cdot p(n) = 2 \quad (2.3)$$

The probability of encountering the end of a run at any given pixel is:

$$p = \frac{1}{\bar{\mathcal{L}}} \quad (2.4)$$

and in our case now it is:

$$p = 0.5 \quad (2.5)$$

In a smoother (and less random) binary picture the run-end probability is less than the 0.5 as given in Equation (2.5). Obviously the number of run-ends is the same as the number of runs. A run edge is where a run ends and the next run starts. Table 2.4 gives bit-plane run coding feasibility test over our sample data. In the table p_1 through p_8 are the probabilities of having a run edge in any word for the bit planes 1 through 8, respectively, with bit 1 being the least significant bit and bit 8 the most significant bit. H_{bits} is the average edge entropy per words for the whole image, which is the sum of that for all eight bit-planes. The average edge entropy per run is calculated as (Pratt [1978]):

$$H_{run} = - \frac{p \log p + q \log q}{p} \quad (2.6)$$

where

$$q = 1 - p. \quad (2.7)$$

Equations (2.6) and (2.4) show that the average edge entropy per message word is:

$$H_{bit} = H_{run} / \bar{\mathcal{L}} = - [p \log p + q \log q] \quad (2.8)$$

Not surprisingly, because the model used here assumes that each pixel's value is independent of its previous pixel's, the H_{bit} value given by equation (2.8) is the same as the information of a binary system having probability of p to assume one of its two possible values (Shannon [1948]). The quantity H_{bits} as listed in Table 2.4 is the sum of all the H_{bit} values calculated for each different bit planes.

Comparing Table 2.4 with Table 1.1 we see that

Table 2.4 Performance Estimation -- Bit-Plane Run Coding

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
p_1	.473	.494	.359	.449	.450	.445	.440
p_2	.472	.546	.363	.459	.465	.465	.468
p_3	.470	.435	.365	.423	.457	.396	.461
p_4	.458	.268	.391	.317	.362	.245	.391
p_5	.387	.172	.290	.201	.221	.169	.282
p_6	.250	.091	.151	.130	.131	.066	.179
p_7	.124	.055	.081	.084	.067	.033	.077
p_8	.051	.014	.028	.037	.023	.014	.059
H_{bits}	6.59	5.33	5.87	5.80	5.76	5.01	6.20

$$H_d > H_{bits} > H_\delta. \quad (2.9)$$

The first part of this inequality is a blessing, while the second part spells out the limitations.

2.4.2 Gray Code

Grim [1980] suggested a revision to the straight bit-plane scheme for run-coding. Instead of using the natural bit-planes of the images, the so-called Gray Code transformation are generated by simple bit-wise operations among the neighboring bit-planes. Mathematically, for an image of n bits, the Gray Code transformation is:

$$G_i = \begin{cases} B_1 & i = 1 \\ B_i \oplus B_{i-1} & i > 1 \end{cases} \quad (2.10)$$

where B_i are the bit plane values, G_i are the corresponding Gray Codes, and $\oplus$ denotes modulo 2 addition, or *exclusive or* operation. The Gray Code transformation is reversible:

$$B_i = \begin{cases} G_1 & i = 1 \\ G_i \oplus B_{i-1} & i > 1 \end{cases} \quad (2.11)$$

The Gray Code test results are shown in Table 2.5. In this table the quantities are defines as the same as that in the Bit-Plane Run Coding.

Table 2.5 Performance Estimation -- Gray Code Run Coding

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
p_1	.473	.494	.359	.449	.450	.445	.440
p_2	.471	.547	.364	.460	.467	.463	.466
p_3	.472	.433	.365	.422	.460	.395	.461
p_4	.459	.266	.396	.320	.361	.244	.392
p_5	.386	.141	.306	.254	.262	.132	.288
p_6	.256	.101	.164	.143	.140	.077	.197
p_7	.136	.055	.077	.084	.069	.036	.121
p_8	.074	.041	.055	.051	.045	.020	.018
H_{bits}	6.73	5.43	6.03	5.99	5.97	5.08	6.19

The Gray Code is said to be able to segregate the random noise in the picture from the actual image, with no theory available. But at least in our evaluation the Gray Code scheme is not successful. On most images the compression ratios are lower than that achievable by the straight bit-plane run coding. With the only exception of Demo4, on which the Gray-Code did achieve a larger compression ratio than that from the bit-plane scheme, the gain is hardly justifiable for the extra computation required.

2.4.3 Alternative Run Code

Lo and Huang [1985] proposed an alternative way to do Run Coding, in which only a

small portion of the messages are coded. Instead of coding every message in the input, they coded only runs with a length larger than 3, and very few other message words which happen to assume a particular escape word value. This scheme prevents the worst scenario case in which the run-length encoder actually creates the code so inefficiently that the total code length can be longer than the original message length. This coding method is a VV scheme, and is instantaneous at the encoder and decoder. Lacking a better name, I refer to this scheme as Alternative Run Code.

A revised version of this alternative run coding can be implemented as the following. First the message code with the least probability is found, and this message is used as the escape code for this particular image. When a message not belonging to a run is encountered, it is transmitted as is, unless its value is the same as the escape code. When a message with the escape code value is encountered, it is transmitted in a pair of code words: the escape code and a 0. When a run is encountered, it is first broken into pseudo runs if it is longer than that 255; each pseudo run, unless its length is less than three, is transmitted by three codes: the escape code, a length code, and a code specifying the gray level of pixels in the run. Pseudo runs shorter than three words long are transmitted *un*-coded, unless the value is the same as the escape code. The end-of-file mark is coded as the escape code and a 1.

Table 2.6 gives the results of this alternative run-coding. In Table 2.6 N_1 is the message words that are not part of a run (or one can say their run-length is unity), N_R is the total number of runs (including the unity length runs), $avg(words)$ is the average code words used per message word, and $avg(bits)$ is the average bits per message word. The value of $avg(bits)$ is 8 times the value of $avg(words)$, since the image pixels are 8 bit wide.

If we compare the N_1 values in Table 2.6 with the N values in Table 1.1 we can find

Table 2.6 Performance Evaluation -- Alternative Run Coding

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
N_1	54071	186583	164012	189841	201253	170602	207863
N_R	57086	214550	172709	214124	222068	204069	225623
$avg(words)$	.971	.995	.743	.924	.935	.929	.935
$avg(bits)$	7.77	7.96	5.94	7.40	7.48	7.43	7.48

that the majority of the pixels do not belong to a run (or they are unity length runs). This makes the total number of runs (N_R) to be near the N value, which in turn makes the alternative run-coding rather inefficient. These same argument makes Contour Encoding inefficient too. Refer to Section 1.2 for Contour Encoding algorithm.

As described above, the coding scheme is a VV scheme with fixed code word length, so the decoding process is simple and fast. But among all the compression algorithms evaluated here the alternative run coding is the least efficient in terms of the compression ratios. Coupled with the fact that the encoder of this algorithm is not instantaneous, it is only natural to say that this implementation fails to have practical value.

Chapter 3

The Algorithm Using the Classifying-Sequencing Techniques

In this chapter a new error-free gray-level image compression algorithm is introduced. It utilizes only integer arithmetic operations, with no requirement on the knowledge of the statistics of the image file, while obtaining compression ratios comparable to those achieved by the Huffman Coding method.

3.1 Overview

The algorithm presented here is designed to preserve the original data in their entirety, to record the data in fewer bits than the original, and to record the compressed result such that the decoder knows what word length to expect at all times. The compression is accomplished through a special classifying and sequencing technique. The algorithm processes the incoming data in three distinct stages: (1) it takes the differences of neighboring data pixels and classifies them; (2) it forms sequences out of these differences; and (3) it codes these sequences.

An image file differs from a general data file in that it exhibits a high correlation between subsequent words, and this correlation is such that the subsequent words are more likely to be close in value than different. It is the continuity of an image that allows us to comprehend it. If an image had no continuity, we would be overwhelmed by the vast amount of information conveyed, perceiving something like the random noise, or "snow", on a television screen after a broadcast. Our brains would shutdown, unable to comprehend the picture. In technical terms, we expect the entropy of the neighboring pixel differences to be lower than the entropy of the data pixels. This fact is used in most compression schemes, and is the starting point for our method as well.

Apparent correlations in an image exist not only between neighboring pixels, but between subsequent neighboring pixel differences as well. This is true also for higher orders of differences. It may not be feasible to calculate and code a higher order difference, however, since doing so would drastically increase the amount of calculation required, while the compression ratios may actually decrease. It is shown in Table 1.1 that the entropy of the second order neighboring pixel differences is on average higher than that of the first order differences. This is because that the effect of increased range of possible difference values overcomes the effect of the added longer range differences.

The scheme introduced here utilizes this correlation between the subsequent (first order) differences in an innovative way. Instead of calculating the second order differences, that is instead of determining the differences between the differences, the algorithm simply assumes that the subsequent differences might, with high probability, be close in magnitude, though their signs may differ. More precisely, if a given difference has a certain value, the next difference may have a different sign, but its absolute value is likely to be close to that of the previous difference value. This can be rationalized as that in most images we prefer that the number of pixels inside of objects to be more than the number of pixels that form the edges of the objects, and that for most objects (or at least for parts of the objects) the gray-levels of the pixels which form them are bounded in a range, hence the neighboring pixel difference values. The range of these gray-levels of an object is partially determined by the texture of this object, and the ranges of different objects may differ because of their texture differences.

With these assumptions, the system classifies the differences according to the range of the difference value. The class of a given difference value is usually the same as the minimum number of bits needed to record it. The number of classes used to record an image is the same as the number of bits per pixel in the image. Each class contains the full

range of the classes below it. For an n bit-per-word image, class 0 contains only null values (i.e. neighboring pixels are equal), class 2 contains -1 to 1 , class 3 contains -3 to 3 , up to class n containing all possible difference values. Class 1 is not used for images of any word length. The class definition and the classifying process are described in Section 3.2.

Once the difference values between neighboring image pixels are computed and classified, the differences are grouped together into sequences. A sequence is a series of subsequent differences belonging to a single class. Chances are, then, some of the sequences will be rather short, which means that proportionally more bits are needed as the overhead load for coding the sequence. It is of vital importance, therefore, to promote the classes of some short sequences, and then merge them to their neighboring sequences. The details of the sequence forming and merging processes are seen in Section 3.3.

After the sequences are formed and necessary mergers executed, the process to code the sequences is relatively simple. The code series of each sequence begins with a header word to specify its class. Then follows the sequence itself, coded according to its class. Classes are divided into three categories, each having a unique coding scheme. The process to code the sequences is described in Section 3.4.

While seemingly complex, the system described above yields high compression ratios and performs reasonably fast. This system is a VV scheme, with code word lengths always predictable by the decoder. It is semi-instantaneous to the encoder, and is instantaneous to the decoder. Hence the decoder is both simpler and faster than the encoder. Experimental results tested on the standard sample files are given in Section 3.5. Results on other forms of images are given in Section 3.6.

3.2 The Difference Classifying Process

After taking the differences of successive pixel elements, the encoder classifies these

differences. The classifying process groups the incoming data items according to the difference values, such that these difference values can be recorded later with the fewest bits possible. Within each class the difference values are to be recorded with code words of fixed length.

For 8-bit images, the neighboring pixel differences are classified as in Table 3.1. There are eight classes for 8 bit-per-word images. Generally, there are as many classes for a type of images as there are bits-per-pixel in these images. For example, 12 classes are used to encode a 12 bit-per-pixel image, which has 4096 gray levels.

Table 3.1 Classes of 8-Bit Images

c	δ_{min}	δ_{max}	2^{c-1}
0	0	0	0.5
2	-1	1	2
3	-3	3	4
4	-7	7	8
5	-15	15	16
6	-31	31	32
7	-63	63	64
8	-255	255	128

In Table 3.1, c is the class number, δ_{min} and δ_{max} are the minimum and maximum difference values allowed in each class, respectively, and the values of quantity 2^{c-1} are to be used for comparison use later.

The first class we consider, class 0, is a special class. It contains differences of value 0 only. This is true for images of any word length. The reason for treating the value 0 specially is that it is highly probable that 0-values appear in consecutive differences. As a

matter of fact, the consecutive image pixels which have identical values are generally referred to as runs in the Run-Coding algorithm. Run in an image may result when imaging a flat background, an object of uniform radiance (e.g. water in LANDSAT images), an under-illuminated area (e.g. background in space and astronomical images), or an area which is so bright that the detector is saturated. The appearance of these runs contributes positively to the overall performance of this compression scheme.

The highest class, in this case class 8, is also a special class. It contains all possible values the differences can assume. This class is special because that once a difference value is outside of the range of the previous class (class 7 in our example), the word length to code the difference is the same or longer than that in the original data, and therefore the original data are coded instead of the difference values. I refer to this class as the full-length class, since the code word length of this class is the same as in the original message word length.

Between class 0 and the full-length class are the “regular” classes. The majority of difference values fall into these classes. As we can see from Table 3.1, the ranges of values these classes contain obey this inequality:

$$|\delta| < 2^{c-1}. \quad (3.1)$$

where δ is a difference value which belongs to this class, and c is the class number. Roughly speaking, the class number c is the bits necessary to record an image element in the respective class. As mentioned earlier, all values in a certain class are to be recorded with equal word length. For example, 5 bits will be needed to record a difference value from -15 to +15, so class 5 contains numbers of this range.

It can be seen that a 5 bit word can record 32 different values, while inequality (3.1) and Table 3.1 provide only 31 different values for this class. We don't fill up this gap with an uneven range, say from -16 to +15, because we need to reserve one value for the stop

word in a sequence (see Section 3.4 for details).

It is for this reason that class 1 is not used. Had we not required that each class reserve one value for the stop word, class 1 could exist, and contain two values. With one of the two values restricted for use as the stop word, however, class 1 can have only one value. But we put this value, namely 0, in class 0. This is because the null value is the only value allowed in that class, thus we do not need to differentiate it from any other value, and the word length needed to record it is 0.

3.3 Sequence Forming and Merging

Sequences are consecutive pixels which belong to the same class. The class of a sequence is the same as the class of the pixels in the sequence. Sequences are coded separately, according to their classes. Initially, consecutive pixels which happen to belong to the same class are grouped together to form sequences. Then some short sequences are merged to their neighbors to reduce the overhead load in the encoding process, increasing the overall compression ratio.

I use the term compression relation to express the equation linking the number of bits of the original message words to the number of bits of the code series. It is shown in Section 3.4 that the compression relation of a sequence, provided that the sequence is not very long, is given by:

$$L = \begin{cases} 3 + c \cdot (S + 1) & c \neq 0 \\ 1 & c = 0 \end{cases} \quad (3.2)$$

where c is the class of the sequence, S is the length of the sequence in *words*, and L is the compressed sequence length in *bits*. The constants 3 and 1 are the overhead burdens. The number 3 is specific for 8 bit images. The overhead is relatively large for short sequences,

and therefore the merging of some short sequences into their adjacent higher-class ones may provide a shorter total compressed code length. Such a merger is accomplished by promoting the short sequence which has the lower class number to the class of the higher-class neighboring sequence. As seen earlier in Table 3.1, each higher class contains the full ranges of the lower classes, therefore promoting a lower class sequence only requires more bits to record each data item, but does not lose any information.

Suppose we have two adjacent sequences. We can use c_1 and c_2 to denote their class numbers, and S_1 and S_2 their sequence lengths. We also use c and S to represent the possible merged sequences. The relationships of the sequence parameters before and after the merger are:

$$c = \max(c_1, c_2), \quad (3.3)$$

$$S = S_1 + S_2, \quad (3.4)$$

Without the merger, we need to use $L_1 + L_2$ bits to record the two sequences, where both L_1 and L_2 are calculated from Equation (3.2) with the parameters for the two sequences. With the merger, we need to use L bits to record the merged sequence, also calculated from Equation (3.2), but with the parameters for the merged sequence. The number of bits saved in the merging process is thus:

$$\Delta L = L_1 + L_2 - L. \quad (3.5)$$

Table 3.2 gives three examples of sequence mergers, with each row representing a separate example.

Substituting the expressions for L_1 and L_2 from equation (3.2), through some derivation, we get the merge merit function:

Table 3.2 Examples of Sequence Mergers

c_1	S_1	c_2	S_2	c	S	ΔL
0	2	2	5	2	7	7
4	1	3	2	4	3	6
2	1	6	5	6	6	1

$$\Delta L = \begin{cases} c_j + 3 - \Delta c \cdot S_j & c_j \neq 0 \\ 11 - \Delta c \cdot S_j & c_j = 0 \end{cases} \quad (3.6)$$

Here Δc is the difference value (absolute value) between the class numbers of the two sequences, and j denotes the sequence which has the lower class number.

In practice the merit function as given in Equation (3.6) is calculated for every pair of adjacent sequences, and mergers are made on those sequence pairs satisfying

$$\Delta L \geq 0. \quad (3.7)$$

The equal condition is included in Equation (3.7) because sometimes mergers of this kind enable subsequent mergers which satisfy $\Delta L > 0$. Mergers of 0 gain also help speed up the decoding process, as the decoder has fewer sequence headers and end words to deal with.

Sometimes it is beneficial to merge three adjacent sequences into one, while the mergers of any adjacent pairs does not provide any gain. Strategies for three, and more than three, sequence mergers can be derived from the two sequence mergers very easily. But again, the law of diminishing returns soon takes over, and one may choose not to evaluate mergers involving more than three adjacent sequences. Detailed discussion on this subject is given in Section 4.3, with evaluation results from tests on our seven standard sample images.

3.4 Coding of the Sequences

In this section we present the different code series for the various sequences. As mentioned earlier, the sequences are grouped into three categories and encoded according to those categories. Compression relations, which are the relations between the compressed code series length and the sequence length, are given for the three categories of sequences. We use 8-bit images as a general example, especially for expressing the compression relations. But the encoding process can still be applied to images of any word length, and the compression relations can easily be revised accordingly.

The code series of a sequence always begins with a header word, indicating its class. For an 8-bit image, the header word is three bits long, since there are 8 possible classes. For 6-bit images, the header still has to be 3 bits long, while for 12-bit images, the header word has to be 4 bits long.

In the discussion below we use *trunc* as the operator for truncation, which truncates the result to an integer value, rather than rounding the result to the nearest integer.

3.4.1 Coding of Class 0 Sequences

A class 0 sequence represents several consecutive pixels which have the same value. In Run Coding terminology the same sequence is called a run. The number of pixels in the sequence is the only information needs to be recorded in coding a class 0 sequence. The code series for a class 0 sequence is:

$$HS \tag{3.8}$$

where H is the header word for class 0, S the length of this sequence (here it is the same as run length). If we choose to use 8 bits to record the sequence length, the maximum length can be recorded is 256 (word lengths of less than 8 bits for the length word have been tried in practice, and the compression results show little difference). When a sequence is longer

than 256, it is broken into two or more pseudo sequences, each containing 256 elements or less.

Since each of the broken off sequences are recorded with 3 bits of header word, and 8 bits of length word, the compression relation for the complete sequence is therefore:

$$L = (3 + 8) \cdot [\text{trunc}(S / 256) + 1]. \quad (3.9)$$

Readers may notice the similarity of the class 0 sequence coding with the run length coding. This is obvious since neither codes the values inside the sequence (or run). The two are different in that the run length coding needs to record the magnitude of the run, while the class 0 sequence coding needs to code its class number. Also, because the class 0 sequences are designed to record only the long sequences, it's only reasonable to use a relatively long word length for the sequence length (I use 8 here as an example). A short class 0 sequence should have been merged to it's neighboring sequences.

3.4.2 Coding of Regular Class Sequences

The code series for a class c sequence, class c being a regular class, is:

$$H\Delta_1\Delta_2\Delta_3 \dots \Delta_S E \quad (3.10)$$

where H is the header word for class c , S the sequence length. The Δ 's record the information about the corresponding difference values, and E is the stop word for class c . The word length for each Δ and for E is c bits. A variety of methods can be used to record the Δ 's. I choose a biased scheme to record the difference values, which is:

$$\Delta_i = \delta_i + 2^{c-1}. \quad (3.11)$$

where δ 's are the difference values. Substituting Equation (3.11) into Inequality (3.1), we get:

$$0 < \Delta < 2^c \quad (3.12)$$

Thus, the possible choice of $\Delta=0$ is used as the stop word.

Because we need 3 bits to record the header word, and c bits to record each Δ value and E , the the compression relation for these classes is thus:

$$L = 3 + c \cdot (S + 1). \quad (3.12)$$

3.4.3 Coding of Full-Length Class Sequences

Full-length class uses the same number of bits to code each pixel as in the uncoded format, and it needs to code every possible pixel value. Therefore its recorded format needs to specify the length of the sequence, because no words can be reserved for a stop word. The code series for a full-length class sequence is:

$$H S d_1 d_2 \dots d_S, \quad (3.13)$$

where H is the header word for class 8 sequences, S the length of this sequence, and the d 's are the original data (*not* the difference values) for the corresponding pixels. The code series for this category is very similar to that of class 0 sequences. Again, if we choose to use 8 bits to record the sequence length, the maximum length that can be recorded is 256. This is sufficient for most class 8 sequences. When a sequence is longer than 256, it is broken into shorter sequences, each with a length less than or equal to 256.

Each of the broken off sequences are recorded with 3 bits of header word, 8 bits of length word, and $8S$ bits of actual data. The compression relation, therefore, is:

$$L = (3 + 8) \cdot [\text{trunc}(S / 256) + 1] + 8S \quad (3.14)$$

Very likely S is small, thus $\text{trunc}(S / 256)$ vanishes, and equation (3.14) simplifies to the same form as equation (3.12). Similarly, when S is small, equation (3.9) yields a constant of 11, and thus equations (3.9), (3.12) and (3.14) can be merged into equation (3.2).

3.5 Experimental Results

Every difference encoding algorithm can be applied in linescan mode, intraframe mode, and interframe (or interband) mode. Programs based on this algorithm have been developed in linescan mode and in intraframe mode, and a interframe mode program set can be essentially the same as the one for the intraframe mode. Table 3.3 presents results of 8-bit image coding using this algorithm in intraframe mode, tested on our standard sample files. Assuming that the computer native word (or *byte*) length is 8 bits, and that each image pixel is stored in such a computer word, the programs are written such that no internal structure of the file is needed. This feature allows us to compress the image files, originally recorded in Data Translation image format, complete with their headers. The Data Translation image file headers occupy a fixed length of 512 bytes.

Table 3.3 Performance Evaluation -- Classifying-Sequencing Method

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
<i>Avg</i>	5.55	4.06	4.22	4.77	4.71	3.88	5.07
t_c	7	26	24	29	28	27	28
t_d	4	14	13	15	15	15	15

In Table 3.3 the units of *Avg* values are bits. The time required to perform the compression and decompression are given by t_c and t_d , respectively. The units are seconds.

It should be pointed out that the sequence merging process involves some trade-offs in the implementation of this algorithm, and it is one of the directions for further research. Detailed implementational considerations leading to the *Avg* values in Table 3.3 are explained in Chapter 4. The results can be compared with the difference entropy H_δ values

in Table 1.1. They can also be compared with the several known algorithms evaluated in Chapter 2, the performance evaluations or estimations of which are given in Tables 2.1 through 2.6.

The speed tests are dependent on the computer used, of course. The numbers given are achieved on a PC-compatible 386SX machine running at 16 MHz. The programs are written in C language, and do not make use of math co-processors. The data buffer used in the encoder is 1024 pixels in size (refer to Section 4.1 on the use of data buffer in the encoder).

3.6 Discussion

As stated earlier, the evaluation of an error-free image compression algorithm can only be sensible if it is compared with some standard measure on the sample images, such as the neighboring pixel difference entropy H_δ and the compression results of the Huffman Code. For our set of sample files, the H_δ values are given in Table 1.1 and the Huffman Code compression results (*Avg*) are given in Table 2.1. On average the Classifying-Sequencing method performs about the same as the Huffman Code. The bits-per-word results of the Classifying-Sequencing algorithm is also close to the H_δ values, and at times even smaller than the H_δ values.

The H_δ values given in Table 1.1 are those of the neighboring pixel differences. The Huffman coding system, as well as the *B*-Codes and *S*-Codes, codes each neighboring pixels separately, thus H_δ values are the theoretical limit that the system performance can approach but never exceed. But H_δ are not entropies of the sequences. For the classifying-sequencing algorithm, these entropy values cannot serve as the performance limit.

As mentioned earlier, the high compression efficiency of this algorithm can be attributed partly to the efficiency of coding class 0 sequences (or runs). The high efficiency

of our coding method can also be partly attributed to an adaptive feature. With most images, the histogram of the neighboring pixel difference values should be a bell-shaped symmetrical curve, peaked at the 0 value. When H_δ is small, we expect this curve to be sharply peaked near the 0 value. When H_δ is large, the curve is more spread out. With our coding method, the values near 0 are recorded by lower class sequences, which use less bits to record each difference value, and generally have smaller overhead. With higher classes, more bits are used to record the increased information.

In principle, the Classifying-Sequencing Code can be more efficient than the Huffman Code, because code words (when sequence header words are ignored) can contain each other as prefix. The key then is how well different code words can share their overheads. These include not only the header words, but also the stop words and length words. From Equation (3.2) we see that for the regular and full-length classes:

$$L = 3 + c \cdot (S + 1) \quad (3.15)$$

If and only if

$$S \gg 1, \quad (3.16)$$

we get

$$c = L / S. \quad (3.17)$$

Which suggests that when sequences are long, a class c sequence indeed uses c bits to code each pixel. But in reality, even after the sequence merging process, in which the class numbers of some of the pixels are promoted, the sequences are not very long, and Condition (3.16) is not always satisfied. The distribution of the sequence lengths does change significantly from image to image.

As mentioned before, class 0 sequences are runs in Run Coding. For an 8 bit image, runs are highly improbable if the image is random. We can calculate the average run length as (non-runs are included as runs with unity lengths):

$$\begin{aligned}\bar{\mathcal{L}} &= \frac{255}{256} \cdot 1 + \frac{1}{256} \cdot \frac{255}{256} \cdot 2 + \left[\frac{1}{256} \right]^2 \cdot \frac{255}{256} \cdot 3 + \dots \\ &= \frac{256}{255} = 1.0039\end{aligned}\quad (3.18)$$

and the average run length for pure runs (runs with length larger than 1) is:

$$\bar{\mathcal{L}}(\mathcal{L} > 1) = \left[\frac{256}{255} - \frac{255}{256} \right] \cdot \frac{256}{1} = 2.0039 \quad (3.19)$$

In reality, runs occur at these conditions: where the imager is saturated, the object is under-illuminated, the scene is uniform, or in parts of the file header if it is not fully used. From the N_1 and N_R values given in Table 2.6 we can find that for the seven image files we use the average run lengths corresponding to the theoretical values given by Equations (3.18) and (3.19) are:

$$\bar{\mathcal{L}} = \frac{N}{N_R} = 1.2258 \quad (3.20)$$

$$\bar{\mathcal{L}}(\mathcal{L} > 1) = \frac{N - N_1}{N_R - N_1} = 3.1757 \quad (3.21)$$

These numbers show that the images differ remarkably from random images, as we expect. A complete theoretical explanation to the efficiency of the Classifying-Sequencing algorithm should include a measure on the long range correlations that could affect Condition (3.16) and the runs as reflected by values given in Equations (3.20) and (3.21). The establishment of this kind of measure may also lead to an optimizing method for the sequence merging process.

Chapter 4

Revisions to the Classifying-Sequencing Algorithm

Since the construction of the basic Classifying-Sequencing Algorithm, many attempts have been made to revise it, to get higher compression ratios than those provided by the simplest implementation of this algorithm. Some of the implementational considerations covered in Sections 4.1 through 4.3 have been incorporated to get the evaluation results as quoted in Table 3.3. Sections 4.4 and 4.5 provide several revision attempts that are not very successful. The different revisions as introduced in Sections 4.3 and 4.5 can be made by only revising the encoder while keeping the decoder unchanged. When compressed images are distributed to many end users and are decoded by these end users before use, the possibility of keeping the decoder unchanged becomes very advantageous.

4.1 Floating-Buffer Technique

The most straightforward implementation of this algorithm is in the linescan mode. In this mode the image data is processed a line at a time. When the code series of one linescan ended before a computer word is finished, this word can be padded with 0's. The code series of the next linescan starts from a new word, just as the first line. The possible file header, or any other form of formatting words, are not coded. The drawbacks of this implementation are that the process is relatively inefficient, and that the encoder and decoder have to have the knowledge of the format of the image files.

The evaluation of the Classifying-Sequencing method, which resulted in the numbers given in Table 3.3, is conducted with a floating-buffer technique. Under this variation of the implementation, the input data file, together with the header, is treated like a stream of one-dimensional data words. The encoder takes in as many words as necessary to fill up an

internal buffer, unless the end of file is encountered before this buffer is filled. Therefore, like the implementation in linescan mode, this implementation is also semi-instantaneous at the encoder and instantaneous at the decoder.

Under the floating-buffer implementation, the classifying and sequencing procedures are applied as if the buffer is a scan line. But in the sequence coding process, the last sequence in the buffer is neither coded nor transmitted. Instead, this sequence is moved to the beginning of the buffer, and the rest of the buffer is then filled with data from the input file. In this way, a natural sequence that is broken off at the end of the buffer can be reconnected when the new data in the buffer are processed.

This method provides easier evaluation of the algorithm, because line-breaks are now ignored, and the internal structures of the data file are no longer important. The file structures may include the header format, the header length, the number of words in a line and the number of lines in an image.

A small revision to the algorithm should be made when using the floating-buffer technique. Because the decoder will not be able to judge how large the image being decoded is, it should facilitate the decoder if an end-of-file marker is written into the compressed data. One possible way of coding this end-of-file marker is coding it as a 0 length full-length class sequence. By doing so, we admittedly reduce the longest possible sequence in this class by one. This should not have any visible effect on the compression ratios, though, since the probability of a full-length class sequence having a length of exactly the multiple of a certain number is very small.

The performance evaluations for the Classifying-Sequencing method as shown in Chapter 3 are conducted using this revision. So will be the evaluations in the following discussions.

4.2 Choosing a Sequence Merging Strategy

While the sequence merging candidates are scarce for 3-sequence mergers (and mergers involving more sequences), two-sequence mergers are performed on many more original sequences, and these mergers play a very important role in the whole algorithm. To get the most compression out of these mergers, we scan the data window many times, setting a high merit function threshold for the mergers at the beginning, and then gradually decrease this threshold until $\Delta L = 0$. Only mergers with a merit function larger than this threshold are performed, while other possible mergers are delayed till the threshold is decreased below their merit functions. This is because if we do not conduct the high-gain sequence mergers first, other mergers with less benefits might actually block them. An example is shown in Figure 4-1.

Figure 4-1 An Example of Effects of Different Orders for Sequence Mergers

In Figure 4-1 each small block represent a sequence. The number in front of the semicolon is its class number c , and the number after the semicolon is its sequence length S , in units of words. A node indicates a possible merger, with the merge merit function given

in the circle beside it.

From the example given in Figure 4-1 we can see that if the mergers are not carried out according to their merit function values, the low-gain merger between the first two sequences would block the possible higher-gain merger between the second and third sequences. Were the mergers executed in their natural orders down the data stream, the first two sequences in the example given in Figure 4-1 would be merged first, then the resultant sequence would be merged with the class 0 sequence that follows, as illustrated by the left branch of Figure 4-1. In contrary, if the higher-gain merger of the second and third sequences are carried out first, then the resultant sequence will not be merged with the class 3 sequence preceding it, as illustrated by the right branch of Figure 4-1.

The decrement step of this benefit function threshold, however, does not have to be 1 for each scan. Test results show that changing this decrement step value from 1 to 2 will not bring significant changes to the compression ratios. Furthermore, when there is change because of the increased step size, the change can be either positive or negative, depending on the actual data. See Table 4.1 for comparisons.

In Table 4.1 Avg_{C-S-2} is the average bits of code used per message word, with up to two-sequence mergers implemented. Similarly, Avg_{C-S-3} and Avg_{C-S-4} are the average code bits per message word with up to three- and four-sequence mergers implemented, respectively. These implementational differences in the encoder are explained in detail in the following section.

4.3 Three and More Sequence Mergers

Most of the beneficial sequence mergers happen to each involve only two sequences. The quantity Avg_{C-S-2} in Table 4.1 reveals the efficiency of the Classifying-Sequencing

Table 4.1 Performance Evaluation

-- The Classifying-Sequencing Method, with Different Implementations

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
Decrement by 1							
Avg_{C-S-2}	5.60	4.10	4.28	4.84	4.77	3.96	5.14
Avg_{C-S-3}	5.55	4.05	4.23	4.77	4.70	3.88	5.07
Avg_{C-S-4}	5.55	4.05	4.23	4.77	4.70	3.88	5.07
Decrement by 2							
Avg_{C-S-2}	5.60	4.10	4.26	4.84	4.78	3.96	5.14
Avg_{C-S-3}	5.55	4.06	4.22	4.77	4.71	3.88	5.07
Avg_{C-S-4}	5.55	4.06	4.22	4.77	4.71	3.88	5.07

method with only two-sequence mergers. However, after the merit function (3.6) is checked with every pair of adjacent sequences and all pairs satisfying (3.7) are merged, there is still room for improvement. One of the things that can be improved on is sequence mergers involving three or more sequences at one time.

It is only after all the two-sequence mergers have been evaluated that we begin to evaluate possible three-sequence mergers. This is because evaluating three-sequence mergers is more complex than evaluating two-sequence mergers. Nevertheless, it is still worthwhile to do, as many adjacent sequence trios will yield beneficial mergers while neither of the adjacent pairs in it will.

Table 4.2 gives three examples of beneficial three-sequence mergers. In this table c_1 , c_2 and c_3 are the class numbers of three adjacent sequences, and S_1 , S_2 and S_3 are the sequence lengths of these sequences, respectively. ΔL_2 is the merit function of a merger

Table 4.2 Examples of Three-Sequence Mergers

c_1	S_1	c_2	S_2	c_3	S_3	ΔL_2	$\Delta L_2'$	ΔL_3
8	8	3	2	7	1	-2	-4	5
5	4	4	11	5	9	-4	-4	4
6	9	3	3	6	13	-3	-3	3

between the first two sequences, and $\Delta L_2'$ is the merit function of a merger between the second two sequences, calculated from equation (3.5). ΔL_3 is the merit function of a merger of all three adjacent sequences, and is calculated as:

$$\Delta L_3 = L_1 + L_2 + L_3 - L \quad (4.1)$$

where L is the code length of the resultant sequence from the three-sequence merger, calculated by applying equation (3.2). The class number and sequence length of the resultant sequence are calculated from:

$$c = \max(c_1, c_2, c_3), \quad (4.2)$$

$$S = S_1 + S_2 + S_3. \quad (4.3)$$

As shown above, the equations (4.1), (4.2) and (4.3) are direct extensions of equations (3.5), (3.3) and (3.4), respectively.

It is of interest to note that for most beneficial three-sequence mergers the first and third sequences in the adjacent trio are of the same class, while the second sequence in it has a lower class than the other two (demonstrated by the second and third examples in Table 4.2). We can therefore limit our search for three-sequence merger candidates to under these conditions, which simplifies the search strategy considerable. The quantity Avg_{C-S-3} in Table 4.1 provides the evaluation results of the Classifying-Sequencing method, in which three-sequence mergers of the kind described above are implemented.

Beneficial four-sequence mergers, after two- and three-sequence mergers are done, can still be found. Like the three-sequence mergers, most of the beneficial four-sequence mergers candidates have the first and last sequences belong to the same class, while the two sequences in the middle have lower classes. But the four-sequence mergers candidates are so few that adding this step into the algorithm does not seem to improve the compression ratio significantly, as shown by the quantity Avg_{C-S-4} in Table 4.1. It can also be observed in Table 4.1 that the compression results of the three- and four-sequence merging process are affected by the strategies used in the two-sequence merging process. The performance test results reported in Table 3.3 are based on the Avg_{C-S-3} values given in Table 4.1, with decrement step of 2 for the merit function threshold in performing the two-sequence mergers.

Because of the low performance enhancement by introducing four-sequence mergers, it is unlikely that mergers involving more than four adjacent sequences can bring any significant improvements. Consequently in general practice it is only necessary to implement up to three-sequence mergers. On the other hand, even the three-sequence mergers can be bypassed if the time-efficiency of the encoder is under concern. The choice of different sequence merger strategies, of course, does no influence the decoder.

4.4 The Floating-Class Technique

There is an asymmetry in the original algorithm which tempts for revision the most. If we map the class definitions from the difference domain into the data domain, we find that while the ranges of class 0 and the regular classes are centered at the previous image pixel value (*PrevData* from here on), the full-length class is centered at the center of the full data range. If we use d to represent the current image pixel value, then the condition for d to be in class 0 is:

$$d = PrevData \quad (4.4)$$

For d to be in a regular class c , the condition is derived from equation (3.1):

$$PrevData - 2^{c-1} < d < PrevData + 2^{c-1} \quad (4.5)$$

And d is certain to fall into the class n , the full-length class:

$$0 \leq d < 2^n \quad (4.6)$$

The above observation seems to suggest, at the first glance, that while the definitions of class 0 and the regular classes incorporate the knowledge of the previous data, that of the full-length class on the other hand assumes no knowledge of this nature at all. A closer look into the matter shows that this observation is not fully justified. At least when *PrevData* is at (or close to) the boundary of the data range, we will know that the current pixel data value is bounded by this same boundary. However, the ranges of the regular classes are still centered at *PrevData*, which means that some of the ranges of these classes are beyond the boundary of the data range.

The floating-class method changed the class designation when the *PrevData* is near the boundary of the data range. When *PrevData* is close to the upper boundary of the data range, the classes are shifted down, and vice versa. It is important that any class still contains all the classes below it after all shifts are done. The class shifting mechanism is implemented as changing equation (3.1) into:

$$| \delta - \delta d_c | < 2^{c-1} \quad (4.7)$$

where δd_c is a constant that is dependent on *PrevData* only, allowing the decoder to reliably shift the classes together with the encoder. Experimentally the floating-class technique is implemented in two different ways, described in the following two subsections. Both of these implementations need to change the decoder as well as the encoder.

4.4.1 The Hard Floating-Class Method

One possible way of implementing the floating-class method can be referred to as the hard floating-class method. With this method the boundaries of the range of data values are considered as hard walls. If the range of regular class c falls partly outside of the range of the data values, we add a bias that is just enough so either the lower or higher boundaries of class c and class n (full-length class) align. Mathematically, the class shifting biases are given by:

$$\delta d_c = \begin{cases} 2^n - PrevData - 2^{c-1} & (PrevData + 2^{c-1} > 2^n - 1) \\ 2^{c-1} - 1 - PrevData & (PrevData - 2^{c-1} < 0) \\ 0 & (\text{otherwise}) \end{cases} \tag{4.8}$$

The requirement for every class to contain the ranges of all classes below it is automatically satisfied in this implementation.

Table 4.3 Performance Evaluation -- The Hard Floating-Class Method

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
Avg	5.54	4.06	4.21	4.77	4.70	3.88	5.07

Table 4.3 gives the test results of the hard floating-class method. It is seen that this implementation of the floating-class method improved visibly the compression ratios of three image files out of the seven. But the improvement is fairly small, only at the third decimal place.

4.4.2 The Soft Floating-Class Method

Another method we tried to implement the floating-class technique is with the soft floating-class method, in which the a regular *class'* shifting bias changes from its previous class by a factor of 2. The class shifting biases are computed as:

$$\delta d_c = \begin{cases} (t - PrevData) / 2^{b-c} & (PrevData < t) \\ (2^n - 1 - t - PrevData) / 2^{b-c} & (PrevData > 2^n - 1 - t) \\ 0 & (\text{otherwise}) \end{cases} \quad (4.9)$$

where t and b are predefined parameters: t is a threshold which defines how close the *PrevData* has to come near the boundary before the classes for the pixel to follow will be shifted, and b is a bias specifying how much the classes will be shifted. The requirement for all the classes contain the classes below will then become:

$$|\delta d_2| \leq 1 \quad (4.10)$$

Table 4.4 gives the testing results on our standard sample files. The results show that when the shifting biases are low (b is small), the number of bits per word is not affected by this scheme. But when the shift biases are high enough (b is large), we actually see an increase in the average bits per word.

The explanation for this is that while the symmetry in the probabilities of an image pixel is affected by the *PrevData*, this effect is fairly small. When the bias is low, the impact on the data set is low, and the result is an average bits-per-word that is not changed significantly. When the bias is high, the symmetry of the class design is distorted so much that some of the data ranging between the *PrevData* value and the boundary value closer to *PrevData* are left out of their original lower classes. Examples on our test data is given in Table 4.4.

Table 4.4 Performance Evaluation -- The Soft Floating-Class Method

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
<i>t=16</i>							
<i>b=6</i>	5.55	4.06	4.21	4.77	4.71	3.88	5.07
<i>b=7</i>	5.55	4.06	4.22	4.77	4.71	3.88	5.07
<i>t=32</i>							
<i>b=6</i>	5.56	4.17	4.34	4.78	4.70	3.88	5.07
<i>b=7</i>	5.54	4.07	4.23	4.77	4.71	3.88	5.07
<i>b=8</i>	5.55	4.06	4.22	4.77	4.71	3.88	5.07
<i>t=64</i>							
<i>b=8</i>	5.54	4.13	4.26	4.77	4.71	3.88	5.07
* <i>b=9</i>	5.54	4.06	4.22	4.77	4.70	3.88	5.07
<i>b=10</i>	5.55	4.06	4.22	4.77	4.71	3.88	5.07
<i>t=128</i>							
<i>b=9</i>	5.54	4.26	4.32	4.77	4.86	3.91	5.07
<i>b=10</i>	5.54	4.06	4.24	4.76	4.73	3.88	5.06
* <i>b=11</i>	5.54	4.06	4.22	4.77	4.70	3.88	5.07

There are only two rows in Table 4.4 (marked with *) that provide average code lengths shorter or equal to that of the results achievable without the floating-class method, as given in Table 4.1 (Avg_{C-S-3} row, with decrement of 2 for the merit function threshold). The price to pay for this small increase in compression ratios is two-fold: the encoder and decoder both have to be rewritten, and more computing time is required for both of them to carry out their work. Comparing the pros and cons, the implementation of this kind of revision is not justified.

4.5 Simple Modulo Technique

For n bit-per-word images, the pixel element values (d 's) are in the range of 0 to (2^n-1) . The range of the neighboring pixel differences (δ 's) is from $-(2^{n+1}-2)$ to $(2^{n+1}-2)$. It is therefore possible to use modulo 2^n operations to reduce the class numbers of some of the δ values without causing confusion on the d values being coded. Specifically, instead of coding δ values, we can code the respective δ' values defined as:

$$\delta' = \begin{cases} \delta + 2^n & (\delta < -2^{n-1}-2^{n-2}) \\ \delta - 2^n & (\delta > 2^{n-1}+2^{n-2}) \\ \delta & (\text{otherwise}) \end{cases} \quad (4.11)$$

where the ranges of δ values in the conditions are chosen so that when we set $\delta' = \delta \pm 2^n$, it is necessary that δ is to fall in class n , while δ' is to fall in class $(n-1)$ or lower.

At the decoder, we have

$$d_{n+1} = (\delta'_n + d_n) \bmod 2^n \quad (4.12)$$

where the result of the modulo operation takes the range of 0 to (2^n-1) . If the computer native word the decoder writes out to is also n bits long, the modulo operation is built-in, and the decoder does not need to treat equation (4.12) differently than that derived from equation (1.4). Otherwise, a mask could be used to filter out the unwanted bits:

$$d_{n+1} = (\delta'_n + d_n) \otimes (2^n - 1) \quad (4.13)$$

where $\otimes$ denotes bitwise *and* operation.

Table 4.5 gives the test results of the implementation the simple modulo technique. Because the test is done on IBM PC-compatible platform, where one of the native computer word (byte, in computer words) has the size as the words of the images, the decoder remains unchanged.

It can be seen from Table 4.5 that impact of implementing the simple modulo technique

Table 4.5 Performance Evaluation -- The Simple Modulo Technique

File	Gold	Mark	Steve	Demo1	Demo2	Demo3	Demo4
Avg	5.55	4.06	4.21	4.77	4.71	3.88	5.07

is very small. This is because that the neighboring pixel pairs that make δ' different from δ are scarce.

Chapter 5

Conclusions

First, I set up some guidelines for evaluating error-free image compression algorithms. I point out that the common practice of providing a print of the sample image is not adequate for error-free image compression algorithm evaluations. I suggested that the entropy values of the neighboring pixel differences, and the compression performance values of the Huffman Code, both performed on the same set of sample images as for the algorithm being evaluated, be given as the minimum set of guides for evaluation.

Several known arts in the field of error-free gray-level image compression are described in Section 1.3. Some of these, including the Huffman Code, are test evaluated with the seven sample files. The evaluations are done in Chapter 2, while the sample files are described in Section 1.4. Table 5.1 gives numerical comparison of the different algorithms evaluated in Chapter 2.

In Table 5.1 Avg are the average bits-per-word of the respective compression algorithms, calculated from the evaluation results given in Tables 2.1 through 2.6, while the H_δ is the entropy of the neighboring pixel differences, calculated from Table 1.1. The averages are performed over all seven sample images, and are weighted by the files sizes. The R_c values are the average compression ratios, which is calculated by dividing 8, the number of bits per word in the original image, by the corresponding Avg values. The result of Arithmetic Coding is given from the theory that the compression results should be arbitrarily close to the entropy value H_δ , as mentioned in Section 1.3. Although the compression ratio of the Arithmetic Coding is higher than the Huffman Code, we can see

Table 5.1 Performance Comparisons

	<i>Avg</i>	<i>R_c</i>
H_δ	4.46	1.79
Huffman	4.49	1.78
B_1	4.92	1.63
B_2	5.11	1.57
B_3	5.43	1.47
S_2	5.70	1.40
S_3	5.18	1.54
S_4	5.07	1.58
Bit-Plane Run*	5.70	1.40
Gray Code*	5.82	1.37
Alternative Run	7.31	1.09
Arithmetic	4.46	1.79
Classifying-Sequencing	4.50	1.78

* Estimated Performance

from Table 5.1 that the difference is small. When not using dynamic modeling, the Arithmetic Coding is more computationally involved than the Huffman Code, and it has the shortcoming that it is more computation intensive at the decoder than at the encoder. Thus I choose to use the Huffman Code as comparison guide.

It can be seen that the most widely discussed Run-Coding method performed the worst among the algorithms evaluated. The B -Codes and S -Codes performed better than the three Run-Coding variations. The performance of the Huffman Coding is the best among the known arts tested. Except for Run-Coding method, the other coding systems discussed all code every message word independently, and their coding results in terms of compression ratios are bounded from above by the value suggested by the message entropy, which in our case is the entropy of the neighboring pixel differences.

A new digital image compression algorithm, using a specific Classifying-Sequencing method, is then proposed. The system codes, for most of the time, the neighboring pixel

differences according to a special classification and sequencing method. For coding first order difference values in images, test results on several sample image files show compression yields approaching the entropy of these difference values, and at times better than this entropy. On average, the compression result of this algorithm is comparable to that provided by the Huffman Code. The performance of this Classifying-Sequencing method is also provided in Table 5.1, which is calculated with values given in Table 3.3.

From Table 5.1 we find that the algorithm using the Classifying-Sequencing method achieves an average compression ratio that is closest to that achieved by the Huffman Code. But there are more than compression ratios in an algorithm that catches our attention. Table 5.2 gives a feature comparison between the Huffman Code and the algorithm using the Classifying-Sequencing method.

Table 5.2 Comparisons Between Huffman Code and the Classifying-Sequencing Code

	Huffman	Classifying-Sequencing
Encoder		
Instantaneousness	Non-inst.	Semi-inst.
Needs to transmit code book	Yes*	No
Uses long integer arithmetic	Yes+	No
Decoder		
Instantaneous	Yes	Yes
Knows next code word length	No	Yes

* Histogram can be transmitted instead

+ Alternatively, floating point arithmetic can be used.

With most digital image compression scenarios, we expect an image to be decoded many times once it is encoded. The *decoding/encoding ratio*, defined as the ratio of the frequencies of the two processes being done on the same images, could be very high for image distribution and archiving, and should be more than unity for other applications.

Thus it is advantageous to shift more burden from decoders to encoders.

The Classifying-Sequencing scheme introduced here satisfies this condition by design. As can be seen from the coding process, the decoder is not only instantaneous, but also knows what word length to expect at all times, thus is very time-efficient. The characteristic that the coding scheme has no parameters which change from image to image makes the decoder efficient also. Because the decoder always has the knowledge of the length of the next code word to follow, and because it does not have a code book that is different from from image to image, it is ensured of a high speed decoding process, which is useful in areas like digital image distribution and archive, where the decoding/encoding ratio is high. In contrast, the Huffman Coding algorithm uses a code book that is different from image to image, and its decoder does not know the word length of the next code word.

The compression efficiency of the Classifying-Sequencing method is relatively high in applications with large uniform areas. This is very helpful in applications such as aerospace, astronomy, remote sensing, and radiology.

Finally, both the encoder and decoder of the Classifying-Sequencing method use only simple logical and arithmetic operations, and without involving long integers or floating point numbers, the algorithm is easy to realize on small and simple processors. The Huffman Coding encoder and decoder have to use long integers to generate and reference to the cook book.

Because the Classifying-Sequencing method codes the input data in a one-dimensional stream manner, it can be applied to general digital data compression for a natural data source.

Possible directions for further developments may include hardware implementations for faster processing speed, and more theoretical analyses on the efficiency and the limitations of this algorithm.

Appendix A
The Seven Sample Images

GOLD

MARK

STEVE

DEMO1

DEMO2

DEMO3

DEMO4

List of References

- Abramson, Norman [1963], "Information Theory and Coding", McGraw-Hill, New York.
- Bauch, Helmut H., Heinz Häberle, Hans G. Musmann, Horst Ohnsorge, Günther A Wengenroth and Hans J. Woite [1974], "Picture Coding", *IEEE Trans. Comm.*, COM-22(9), 1158-1167.
- Chevion, Dan, Ehud D. Karnin and Eugene Walach [1991], "High Efficiency, Multiplication Free Approximation of Arithmetic Coding", Data Compression Conference, Proceedings, April 8-11, 1991, Snowbird, Utah, 43-52.
- Clarke, R. J. [1985], "Transform Coding of Images", Academic Press, London.
- Cleary, J. and I. Witten [1984], "A Comparison of Enumerative and Adaptive Codes", *IEEE Trans. Inform. Theory*, IT-30, 306-315.
- Fano, Robert M. [1949], "The Transmission of Information", Tech. Rep. 65, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge.
- Fano, Robert M. [1961], "Transmission of Information; A Statistical Theory of Communications", MIT Press, New York.
- Golomb, Solomon W. [1966], "Run Length Encodings", *IEEE Trans. Inform. Theory*, IT-12(3), 399-401.
- Gonzalez, Rafael C. and Paul Wintz [1977], "Image Encoding", Chapter 6 in "Digital Image Processing", Addison-Wesley, Reading, Massachusetts.
- Goodall, W. M. [1951], "Television by Pulse Code Modulation", *Bell Syst. Tech. J.*, 30(1), 33-49.
- Grim, Larry B. [1980], "Digital Video Image Compression", *Proc. Micro Delcom '80*, 119-122.
- Kretz, Francis [1975], "Subjectively Optimal Quantization of Pictures", *IEEE Trans. Comm.*, COM-23(11), 1288-1292.
- Haffman, David A. [1952], "A Method for the Construction of Minimum-Redundancy Codes", *Proc. IRE*, 40(9), 1098-1101.
- Laemmel, A. E. [1951], "Coding Processes for Bandwidth Reduction in Picture Transmission," Microwave Research Institute, Polytechnic Institute of Brooklyn, Report R, 246-251.
- Lo, Shih-Chung B. and H. K. Huang [1985], "Error-Free and Irreversible Radiographic Image Compression", *Proc. SPIE*, 536, 170-177.

- Lo, Shih-Chung B., Brian Krasner and Seong K. Mun [1990], "Noise Impact on Error-Free Image Compression", *IEEE Trans. Med. Imaging*, 9(2), 202-206.
- Millard, J. B., and H. I. Maunsell [1970], "Digital Encoding of the Video Signal", *Bell Syst. Tech. J.*, 50(2), 459-479.
- O'neal Jr., J. F. [1966], "Predictive Quantizing Systems (Differential Pulse Code Modulation) for the Transmission of Television Signals", *Bell Syst. Tech. J.*, 45(5), 689-721.
- Pasco, R. [1976], "Source Coding Algorithms for Fast Data Compression", Ph.D. thesis, Dept. of Electrical Engineering, Stanford University, Stanford, California.
- Pennebaker, W. B., J. L. Mitchell, G. G. Langdon, Jr. and R. B. Arps [1988], "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder", *IBM J. Res. Develop.*, 32(6), 717-726.
- Pratt, William K. [1978], "Digital Point Processing Image Coding", Part 6, Chapter 23 in "Digital Image Processing", John Wiley & Sons, New York.
- Rissanen, J. [1976], "Generalized Kraft Inequality and Arithmetic Coding", *IBM J. Res. Dev.*, 20, 198-203.
- Rissanen, J. [1979], "Arithmetic Codings as Number Representations", *Acta Polytech. Scand. Math.*, 31, 44-51.
- Schindler, H. R. [1970], "Delta Modulation", *IEEE Spectrum*, 7 (October), 69-78.
- Schwartz, Jay W. and Richard C. Barker [1966], "Bit-Plane Encoding: A Technique for Source Encoding", *IEEE Trans. Aerospace and Electronic Systems*, AES-2(4), 385-392.
- Shannon, Claude E. [1948], "A Mathematical Theory of Communication," *Bell Sys. Tech. J.*, 27, 379-423 and 623-656.
- Tjalkens, Tjalling J. [1987], "Efficient and Fast Data Compression Codes for Discrete Sources with Memory", doctoral thesis, Technische Hogeschool Eindhoven (Netherlands), republished on microfiche by the US Dept. of Commerce, Nat. Tech. Info. Service, Springfield, Virginia.
- Vaughan-Nichols, Steven J. [1990], "A One-Sweep Approach", *BYTE*, 15(12), 334-335 (inset).
- Welch, Terry A. [1984], "A Technique for High-Performance Data Compression", *Computer*, 17(6), 8-19.
- Witten, Ian H., Radford M. Neal and John G. Cleary [1987], "Arithmetic Coding for Data Compression", *Commun. ACM*, 30(6), 520-540.
- Ziv, Jacob and Abraham Lempel [1977], "A Universal Algorithm for Sequential Data Compression", *IEEE Trans. Inform. Theory*, IT-23(3), 337-343.