Techniques for lossless image compression

Daniel R Jenkins

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

https://digitalscholarship.unlv.edu/rtds/3327

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
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.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600
TECHNIQUES FOR LOSSLESS
IMAGE COMPRESSION

by

Daniel R. Jenkins

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Engineering
University of Nevada, Las Vegas
August 1997
The Thesis of Daniel R. Jenkins for the degree of Master's of Science in Computer Science is approved.

Chairperson, Evangelos A. Yfantis, Ph.D.

Examiner Committee Member, Lawrence L. Larmore, Ph.D.

Examiner Committee Member, John Minor, Ph.D.

Graduate Faculty Representative, Peter Stubberud, Ph.D.

Dean of the Graduate College, Ronald W. Smith Ph.D.

University of Nevada, Las Vegas
August 1997
ABSTRACT

Popular lossless image compression techniques used today belong to the Lempel-Ziv family of encoders. These techniques are generic in nature and do not take full advantage of the two-dimensional correlation of digital image data. They process a one-dimensional stream of data replacing repetitions with smaller codes. *Techniques for Lossless Image Compression* introduces a new model for lossless image compression that consists of two stages: transformation and encoding. Transformation takes advantage of the correlative properties of the data, modifying it in order to maximize the use of encoding techniques. Encoding can be described as replacing data symbols that occur frequently or in repeated groups with codes that are represented in a smaller number of bits. Techniques presented in this thesis include descriptions of Lempel-Ziv encoders in use today as well as several new techniques involving the model of transformation and encoding mentioned previously. Example compression ratios achieved by each technique when applied to a sample set of gray-scale cardiac images are provided for comparison.
LIST OF TABLES

Table 1  Adaptive Huffman Compression Ratios ............................................................. 8
Table 2  LZ77 Compression Ratios .............................................................................. 11
Table 3  PKZIP Compression Ratios ......................................................................... 13
Table 4  LZW Compression Ratios .............................................................................. 17
Table 5  1st Level Sub Sampling Frequencies (frame1.tif) .............................................. 26
Table 6  4th Level Sub Sampling Frequencies (frame1.tif) ............................................. 28
Table 7  4th Level Sub Sampling + Huffman Compression Ratios .................................... 30
Table 8  4th Level Sub Sampling + LZ77 Compression Ratios ....................................... 30
Table 9  4th Level Sub Sampling + LZW Compression Ratios ....................................... 31
Table 10  4th Level Sub Sampling + (3x3) Block Coding Compression Ratios ............... 31
Table 11  4th Level Sub Sampling + (4x4) Block Coding Compression Ratios ............... 32
Table 12  DPCM Frequencies (frame1.tif) ................................................................... 36
Table 13  DPCM + Huffman Compression Ratios ......................................................... 38
Table 14  DPCM + LZ77 Compression Ratios ............................................................... 38
Table 15  DPCM + LZW Compression Ratios ............................................................... 39
Table 16  DPCM + (3x3) Block Coding Compression Ratios .......................................... 39
Table 17  DPCM + (4x4) Block Coding Compression Ratios .......................................... 40
Table 18  1st Level Integer Wavelet Frequencies (frame1.tif) ........................................ 45
Table 19  3rd Level Integer Wavelet Frequencies (frame1.tif) ....................................... 47
Table 20  3rd Level Integer Wavelet + Huffman Compression Ratios ............................ 49
Table 21  3rd Level Integer Wavelet + (3x3) Block Coding Compression Ratios .......... 49
Table 22  3rd Level Integer Wavelet + (4x4) Block Coding Compression Ratios .......... 50
Table 23  1st Level Floating Point Wavelet Frequencies (frame1.tif) ............................. 54
Table 24  4th Level Floating Point Wavelet Frequencies (frame1.tif) ............................ 56
Table 25  4th Level Floating Point Wavelet + Huffman Compression Ratios ................. 58
Table 26  4th Level Floating Point Wavelet + (3x3) Block Coding Compression Ratios ... 58
Table 27  4th Level Floating Point Wavelet + (4x4) Block Coding Compression Ratios ... 59
Table 28  Sample compression ratios summary ............................................................ 61
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Adaptive Huffman Compression Ratios</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>LZ77 Compression Ratios</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>PKZIP Compression Ratios</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>LZW Compression Ratios</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Original image pixel data</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Sub Sample transform data</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>1st Level Sub Sampling Frequencies (frame1.tif)</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>4th Level Sub Sampling Frequencies (frame1.tif)</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>4th Level Sub Sampling Compression Ratios</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>Original image data</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td>DPCM transform data</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>DPCM Frequencies (frame1.tif)</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>DPCM Compression Ratios</td>
<td>40</td>
</tr>
<tr>
<td>14</td>
<td>Original image pixel data</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>Integer Wavelet transform low-pass component data</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>Integer Wavelet transform high-pass component data</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>Integer Wavelet inverse transform data</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>Integer Wavelet vertical low-pass transform data applied to the Y data set</td>
<td>43</td>
</tr>
<tr>
<td>19</td>
<td>Integer Wavelet vertical high-pass transform data applied to the Y data set</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>Integer Wavelet vertical low-pass transform data applied to the Z data set</td>
<td>43</td>
</tr>
<tr>
<td>21</td>
<td>Integer Wavelet vertical high-pass transform data applied to the Z data set</td>
<td>43</td>
</tr>
<tr>
<td>22</td>
<td>Integer Wavelet inverse transform to restore the Y data set</td>
<td>44</td>
</tr>
<tr>
<td>23</td>
<td>Integer Wavelet inverse transform to restore the Z data set</td>
<td>44</td>
</tr>
<tr>
<td>24</td>
<td>1st Level Integer Wavelet Frequencies (frame1.tif)</td>
<td>46</td>
</tr>
<tr>
<td>25</td>
<td>3rd Level Integer Wavelet Frequencies(frame1.tif)</td>
<td>48</td>
</tr>
<tr>
<td>26</td>
<td>3rd Level Integer Wavelet Compression Ratios</td>
<td>50</td>
</tr>
<tr>
<td>27</td>
<td>Original image pixel data</td>
<td>52</td>
</tr>
<tr>
<td>28</td>
<td>Floating Point Wavelet transform low-pass component data</td>
<td>53</td>
</tr>
<tr>
<td>29</td>
<td>Floating Point Wavelet transform high-pass component data</td>
<td>53</td>
</tr>
<tr>
<td>30</td>
<td>Floating Point Wavelet inverse transform data</td>
<td>53</td>
</tr>
<tr>
<td>31</td>
<td>1st Level Floating Point Wavelet Frequencies (frame1.tif)</td>
<td>55</td>
</tr>
<tr>
<td>32</td>
<td>4th Level Floating Point Wavelet Frequencies (frame1.tif)</td>
<td>57</td>
</tr>
<tr>
<td>33</td>
<td>4th Level Floating Point Wavelet Compression Ratios</td>
<td>59</td>
</tr>
<tr>
<td>34</td>
<td>Sample compression ratios summary</td>
<td>61</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Image compression is a topic that has been studied intensely over the past few decades. Because humans rely heavily on their visual senses, complex ideas or concepts can often be communicated easier with images than by other means. This may be the basis for the saying, “a picture is worth a thousand words.” The use of digital images in software to facilitate understanding will continue to increase in proportion to the level of technology. The disadvantages of working with digital images is storage and retrieval. A picture may be worth a thousand words, but when dealing with digital images it can also be worth several thousands of bytes. Today the average computer monitor has a resolution of 800x600 pixels or greater; an uncompressed true-color image (32 bits per pixel) of this size would require over a Megabyte to store on a disk.

Recently, the technology of disk storage has improved so that storing many large binary data objects such as digital images may not be a problem, even for the average personal computer. However, with the advent of the Internet and wide area networks, the practice of transferring large binary data objects efficiently over phone or data lines with limited bandwidth remains a problem. If an image can be compressed at a ratio of 2:1 before being transmitted, it will take half as long to be received. There is a small cost of decompressing on the receiving end, but with today’s processing speed that cost is
negligible. Because the demand for sending and receiving information is increasing, the study of data compression will continue.

Most of the efforts in image compression have been directed towards lossy image compression. Lossy image compression is a compression technique that results in a loss of original image data. An image can be compressed a great deal more by accepting a small loss of the original image quality. For example, the current JPEG (Joint Photographers Expert Group) image format can compress images up to a 20:1 ratio. Newer techniques using Wavelet transforms that can achieve even higher compression ratios that will probably be standard in the future. With compression ratios this high, it is easy to see why so much research has been done in this area.

There are some scientific areas where even a small loss of image quality can be perceived as unacceptable. For example, in the field of remote sensing, where scientists use satellite images to study distant objects, or in the field of medicine, where the integrity of every pixel counts. The sample images used in this thesis are 8-bit gray-scale cardiac images. Even though the use of lossless image compression may not be widespread, the need for it still exists and will be the focus of this thesis.

Popular lossless image compression techniques in use today belong to the Lempel-Ziv (LZ) family of encoders. These techniques are generic in nature, i.e., they were designed to compress any form of data that is repetitious. LZ encoders do not take full advantage of the two-dimensional correlation of digital image data. They perform on a one-dimensional stream of data replacing repetitions with smaller codes. Because LZ encoding does not take advantage of the two-dimensional nature of digital images it is not referred to in this thesis as a "sophisticated" image compression technique. A description
of the LZ encoding schemes and other lossless data compression background information is included in Chapter 2.

Lossy image compression techniques capable of achieving high compression ratios, like that of the JPEG, involve a number of stages. These stages usually fall into three categories: transformation, quantization, and encoding. Transformation is achieved by taking advantage of the correlative properties of the data and modifying it in a way that minimizes the energy of the data. Quantization takes the data from the transform and modifies it by placing it into groups so the range of data values is decreased. Encoding replaces data symbols that occur frequently or in repeated groups with codes that are represented in a smaller number of bits. Of these three compression stages, quantization always results in a loss of data and should not be applied to lossless image compression techniques. In Chapter 3, the transformation and encoding stages of lossy image compression are adopted to introduce a new “sophisticated” model of lossless image compression. Chapter 3 also introduces a new encoding technique for images referred to as Block coding.

In Chapters 4-7 the Sub Sampling, Differential Pulse Code Modulation, Integer Wavelet, and Floating Point Wavelet transforms are described in detail. Compression ratio results of each transform when combined with the encoding techniques described in Chapters 2 and 3 are included as well.

Chapter 8 contains conclusions and suggestions for further research in the area of lossless image compression.
CHAPTER 2

BACKGROUND

Many techniques have been used for lossless data compression in the past. This chapter will concentrate on those that are the most commonly used today. They will be discussed in order of their discovery to give an historical perspective of the lossless data compression field. Each method will be described briefly and some examples of their respective compression ratios will be given as well.

Huffman Coding

A likely place to begin the study of lossless data compression is with Huffman coding. (Huffman, 1952) This algorithm has been studied frequently since it was first published, and is still used in the encoding stage of many compression algorithms today. Various uses of Huffman coding will be discussed later.

The idea behind the Huffman coding algorithm is that, for a given set of data, the symbols occurring more frequently should be stored with fewer bits than those occurring less frequently. The integral part of this algorithm is building the Huffman tree. The Huffman tree is a binary tree containing each symbol of the data set as a leaf of the tree. Once the tree is built, the binary Huffman code for that symbol can be found by traversing the tree from the root; each left node you take adds a 0 bit to the code and each right node

4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
a 1. You could use 1 for left and 0 for right, as long as you are consistent. The algorithm for building the Huffman tree is:
1. Find the frequency of every symbol in the set of data and put it in a list of nodes to be added to the tree.
2. Create a new parent node labeled with a frequency equal to the sum of the two lowest frequency nodes in the list. Set the lower frequency of the two to be the left child and the higher to be the right. (or visa versa, just be consistent)
3. Add the new parent node from step 2 to the list.
4. Remove the two lowest frequency nodes found in step 2 from the list.
5. Repeat steps 2-4 until there is only one node remaining in the list. This is the root of the Huffman tree.

One particular aspect of traditional Huffman algorithm is that the decoder must first be able to reproduce the Huffman tree in order to decode all the Huffman symbols. An efficient method of passing the Huffman tree to the decoder, referred to as “bit packing” is described below.

1. Begin traversing at the root of the tree.
2. Traverse each left branch, setting the next output bit to 0 for each branch taken until you reach a leaf/symbol in the tree.
3. Set the next output bit to 1 and save the symbol to the output stream, in however many bits your symbols would require.
4. Trace back up the tree until the next available right branch can be taken and go there.
5. Repeat steps 2-4 until there are no more right branches to take in step 4; all the symbols will have been output at this point.

The decoder can now rebuild the Huffman tree by a similar reverse process.

1. Create a root node.

2. For every 0 bit in the input stream create a left child from the current node and traverse to it until you reach a 1 bit.

3. Get the current symbol value from the input stream and record it at this node.

4. Trace back up the tree until you reach a right node that has not been created. Create a right node at this point and traverse to it.

5. Repeat steps 2-4 until there are no more right nodes to be created. The tree will be complete at this point.

Adaptive Huffman Coding

A variation of the Huffman coding algorithm can be achieved by making it so that the Huffman tree doesn't need to be passed to the decoder. The encoder and the decoder will build and update the tree as each symbol is processed. This is accomplished by initializing the Huffman tree with a root node and two leaf-code nodes; an escape code to signal a new symbol to be added to the tree, and an end code to signal the end of the data set. The encoding algorithm for this is:

1. Get the next symbol from input.

2. If the symbol is already in the tree, output the code for the symbol and update the Huffman tree for this symbol in the method described below.
3. If the symbol is not in the tree, output the escape code followed by the actual symbol itself, add the new symbol to the tree at the very bottom left of the tree with a frequency of 0, and update the Huffman tree as described below.

When updating the Huffman tree you must begin at the symbol in the tree and increment its frequency by 1 repeating this process for every parent of the symbol. If after incrementing the frequency of the current node you come across a node that has a higher frequency than its next/right sibling, then this node and its children must be swapped with the last node and its children in sibling order with the same frequency of the next/right sibling. The Huffman tree should always satisfy the following two properties:

- **Parent property:** The frequency of the parent is equal to the sum of the frequencies of its children.
- **Sibling property:** The frequency value of the sibling nodes must be in increasing order.

The order of sibling nodes begins at the deepest level of the tree and goes from left-to-right on each level up the tree.

Huffman is a good general coding algorithm and works best on data that has a few symbols with high frequencies of use. The results of the five sample images compressed using only Huffman coding is shown in Table 1 and Figure 1.
Table 1

Adaptive Huffman Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>202522</td>
<td>1.214</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>202769</td>
<td>1.213</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>202774</td>
<td>1.213</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>203064</td>
<td>1.211</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>201466</td>
<td>1.221</td>
</tr>
</tbody>
</table>

Figure 1. Adaptive Huffman Compression Ratios

LZ77 Coding

The LZ77 coding algorithm changed the field of data compression. (Lempel and Ziv, 1977) Until this algorithm was published, research in the field was directed mostly at
variations of Huffman coding. LZ77 introduced the idea of encoding more than one data symbol with a single code, which is quite a bit different than the one-to-one replacement accomplished by Huffman coding.

The LZ77 algorithm is fairly simply described and is often referred to as a "sliding window dictionary" technique. There are two character buffers used in the LZ77 algorithm. A limited look ahead buffer that contains the current series of characters to be encoded and a limited window buffer of those characters that have been previously encoded. It is called a sliding window because as the data is encoded the window buffer slides along the data set until reaching the end. The algorithm replaces as many of the immediate characters in the look ahead buffer with a string of characters seen previously in the window buffer by specifying an offset and length of this previous character string. By specifying this offset and length code combination a string of characters can be replaced up to the maximum length of the look ahead buffer. The size of the look ahead and window buffers are predetermined for each encoding scheme. The capacity for compression is increased by increasing the size of these two buffers however, the size of the code combinations will be increased as well. There is also a minimum match value that needs to be determined that is dependent on the window size and length. Discussion of how to determine these parameters continues after a more formal description of the algorithm below.

1. Find the longest match of the current character string in the look ahead buffer to that in the window buffer.
2. If there is a match longer than the predetermined minimum, set the next output bit to 1
   signifying that this code is to be an offset-length pair and output the offset-length to
   the output string.
3. If there is no minimum match, set the next output bit to a 0 followed by the current
   input character.
4. After the current string or character has been processed, advance the window and look
   ahead buffers to the next input string.
5. Repeat steps 1-4 until all input characters have been processed.

   NOTE: The algorithm described above is actually an improved variant on the one
   published in the 1977 article. The original algorithm output every code as length +
   offset + "followed by" character triple. If a suitable match does not exist in the
   window then the offset and length would both be ‘0’ and it would be followed by
   the current character being processed.

   The next consideration is how the maximum offset, maximum match length, and
   minimum match length values are determined. The greater the maximum values for the
   offset and length, the greater the potential for compression. Remember however, that you
   have to store each of these so there is a break even point when determining the minimum
   match value. For example if the window size is 14 bits and the maximum length is 8 bits,
   then unless there is a match of at least 3 characters then it would take fewer bits to output
   the code for each single character, one at a time. There is a direct relationship between
   the minimum match value and how many matches will be found in a given set of data.
Intuitively, it can be seen that the larger the minimum match value then the number of matches found will be smaller.

Another consideration of the parameters is the speed and implementation of the LZ77 algorithm. Searching a sliding window (especially a large one) for maximum matches can be a difficult task, requiring sophisticated data structures. For the purpose of this thesis, I have implemented the LZ77 algorithm with a 12 bit (4096 bytes) window, a 4 bit (17 bytes) look ahead or maximum replacement length, and a minimum match of 2 characters. The output length is 17 bytes because there are 16 possible values in 4 bits (0-15) and the first value of 0 represents the minimum match length of 2. The compression ratios achieved by this algorithm on the sample files are shown in Table 2 and Figure 2.

Table 2

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>188276</td>
<td>1.306</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>188714</td>
<td>1.303</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>188716</td>
<td>1.303</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>188776</td>
<td>1.303</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>186349</td>
<td>1.319</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The LZ77 algorithm compresses repetitive data efficiently, as can be seen by a simple example of a character string of 10 A's. Initially there is nothing in the window so the first code output is the single character 'A'. The next time, though, it can match the next 9 A's as a single offset-length pair because of the immediate expansion of the sliding window.

Decoding of LZ77 is almost trivial compared to the encoding; there is no sophisticated searching involved. Either a single character is decoded one at a time or a character string seen previously is decoded from the previous output buffer and information given in the offset-length code pair.

A variant of the LZ77 algorithm is used in current popular compression utilities such as PKZIP for DOS and gzip for UNIX. The variation uses Huffman coding on the symbols output by the LZ77 algorithm to provide an additional layer of compression. It
accomplishes this by using 2 separate Huffman trees. One Huffman tree is for the window offsets while the other is for the combined lengths and single characters. The LZ77 codes for a given set of data will have some values that occur more frequently than others. Huffman coding can take advantage of this property and provide additional compression.

Compression ratios achieved by this variant of the LZ77 algorithm can be seen in Table 3 and Figure 3 which show the results of using the PKZIP utility to compress the sample files. The improvement of the compression ratios is considerable compared to those shown previously. Use of the Huffman coding is partly responsible for this. However, another factor is that a much larger sliding window (15 bits, or 32 kilobytes) and a larger match length (8 bits or 258 bytes - 3 bytes minimum match) is implemented in PKZIP.

Table 3

PKZIP Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>160784</td>
<td>1.529</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>161165</td>
<td>1.526</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>161018</td>
<td>1.527</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>161237</td>
<td>1.525</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>159062</td>
<td>1.546</td>
</tr>
</tbody>
</table>
The LZ78 algorithm sparked the next revolution in lossless data compression. (Lempel and Ziv, 1978) LZ78 is similar to LZ77 in the concept of replacing current input strings with those seen previously. The idea presented in LZ78 was to build a code table, initially empty, made up of string + character combinations rather than using a sliding window. Here a string is defined as zero or more characters. Each time a new string + character combination is found, it is added to the code table. This way repeated strings could be referred to with a code + character combination rather than an offset + length pair as in LZ77. The effect of this algorithm is that the string matches are not limited to those found in a restricted sliding window or to a given length. The limiting factor of LZ78 is the size of the code table being built.
Use of the LZ78 algorithm was not wide spread immediately after its publication. Implementation of the algorithm they way it was presented was difficult. It wasn’t until a variant of the LZ78 algorithm, called LZW, that the scientific community at large began to really notice the power of the LZ78 algorithm. (Welch, 1984) Like the variant of the LZ77 algorithm described earlier Welch’s LZW algorithm outputs only the codes representing a string of symbols rather than a code + character combination. The formal description of the LZW encoding algorithm is described below.

1. The string is initialized to the value of the first input character.
2. The character is assigned the value of the next input character.
3. If string + character is found in the code table, the new value of the string is assigned its old value + character.
4. If string + character is not found in the code table, output the code for the string and add string + character to the code table, assigning the new value of the string to the value of the character.
5. Repeat 2-4 until there are no more input characters.
6. Output the code for the string.

The LZW decoding algorithm is not as intuitive as the encoding algorithm, but it is not impossible taken one step at a time as follows:

1. Get the old-code from the input stream and output it immediately. The first code from the compressed file doesn’t need to be decoded because the code table is empty.
2. Get new-code from input stream.
3. If new-code is not in the decoding table, assign the decoded string to be the value of
   the decoded old-code + current character.
4. If new-code is in the decoding table, assign the decoded string to be the decoded new-
   code.
5. Send the decoded string to the output stream
6. Assign the current character to be the value of the first character in the decoded string.
7. Add the old-code + current character to the decoding table.
8. Assign the old-code to the value of the new-code.
9. Repeat steps 2-8 until there are no more codes available in the input stream.

   While the LZW algorithm may be a little more complex than the LZ77, efficient
implementation of the LZW can be achieved more simply. The LZW algorithm does not
directly search for strings. Instead, it searches for code + character combinations stored in
the code table. This search can be implemented efficiently by using a single hash function
with a key based on the value of the code and character. The efficiency of using a hash
function to search for strings is an advantage of the LZW algorithm.

   Use of LZW algorithm can be found in the compression utilities such as UNIX
COMPRESS and DOS ARC. Perhaps the most popular use of the LZW compression
algorithm is in the GIF graphics file format originally developed by CompuServe in 1987.
GIF (pronounced as “JIF”) stands for Graphics Interchange Format. There are several
variations of the LZW algorithm, many involving manipulation of the code table, including
monitoring the compression ratio and flushing the table when it begins to degrade. Such
variations will not be described in this thesis.
Two drawbacks to the LZW algorithm that should be mentioned here. One drawback is that the LZW does not lend itself to the additional Huffman coding component which proved useful in the LZ77. The second, and perhaps most devastating drawback to the LZW algorithm, is the fact that it is patented by the Unisys Corporation and there are legal and financial issues to be considered for it to be used in commercial applications.

Table 4 and Figure 4 provide examples of compression ratios achieved by the LZW algorithm on the sample files. The example implementation uses a 15 bit code table that is not flushed when full.

Table 4

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>179682</td>
<td>1.368</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>180260</td>
<td>1.364</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>179650</td>
<td>1.369</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>182253</td>
<td>1.349</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>180225</td>
<td>1.364</td>
</tr>
</tbody>
</table>
Figure 4. LZW Compression Ratios
CHAPTER 3

COMPRESSION IN TWO STAGES

The compression model described in this thesis has two stages; transformation and encoding. Transformation is a means of changing the original data into a form that can be compressed more effectively by the encoding method in the second stage of the process. In Chapter 2 a few current lossless compression techniques were described in detail. It is important to note that there were no transformations applied to the data prior to these compression algorithms. As mentioned previously, these techniques are intended for generic data that may have some symbols occurring more frequently than others (Huffman) or that is repetitious (LZ variants) in nature. Moderate compression is inherent in each, but they were not specifically designed to compress images. Because of their generic nature they will be used in the encoding stage of our compression model and hereafter referred to as encoding/coding techniques.

Transformation

In order to transform data to a form more readily compressible, the original data set must exhibit certain correlative properties. Digital images contain highly correlated data. For example, the color or intensity of any one pixel in an image is very close to, and often exactly the same as, its neighboring pixels. The transformations described in the
following chapters will take advantage of this correlation by, in general, recording the changes in the pixel values that take place throughout the image. These changes can be represented by small positive and negative numbers with the value of zero being the most frequent. A histogram of the data after a transformation of this sort you would exhibit a peak where the zeros are and sharp downward slopes for increasing positive and negative values. The steeper the peak of the histogram the more effective the encoding stage of the compression process will be.

Another property of our transformations is that they have to be completely reversible. In other words, every transformation must have an inverse that will restore the data to its exact original form, without any loss of data. Otherwise our compression algorithm will not be lossless. In Chapter 7, "AN ALTERNATIVE TO LOSSLESS IMAGE COMPRESSION," a transformation of this sort is described.

Encoding

The best way to see why transformations mentioned previously increase the effectiveness of encoding techniques is to examine the data properties that must exist in order for the encoding techniques to perform well. As described in Chapter 2, Huffman coding works best on data that has a few symbols with a large frequency of occurrence. The histogram describing data after the transformation fits this profile very well. The LZ family of encoding works best on data that is highly repetitive. After the transformation the data will be even more repetitious because of the concentration of small positive/negative numbers. One drawback to the LZ family of encoding, when applied after transformation, is that the data must be kept as unsigned characters instead of
positive and negative integers. As a result the histogram will change in appearance. The peak will still be at zero but the slopes will decline from the values of 1, 2, 3, etc. and 255, 254, 253, etc.

Zero Run Length/Block Encoding

Another encoding scheme used in the image compression techniques used in the following chapters is called “run length” encoding. Run length encoding is useful when you have a series of values that remain the same in a given section of data. Traditional run length encoding schemes have two parts. The first is the repeated symbol and the second is the number of times that symbol repeats. There is a specific case of run length encoding called “zero run length” encoding that will be used as an additional encoding tool. Zero run length encoding means that instead of looking at each data symbol to see if it repeats, only the zeros are coded as a run length and the rest is left alone. As mentioned previously, the transforms described in this thesis will always create data with zero values being the most frequent. Traditional zero run length encoding in images, as with the other encoding techniques, is performed on a scan line. However, keeping in mind that images are two-dimensional objects, zero run lengths will be determined from two-dimensional blocks of data instead. Hereafter we will refer to this technique as “Block” coding. The size or dimensions of a block can vary, but care must be taken not to make them too large or small. Examples 3x3 and 4x4 block coding will be shown in the following chapters.

To implement Block coding, two bitmaps are necessary to represent the transformed data. The first bitmap will describe whether a given block of data is either all zeros with a bit set to ‘0’, or that it contains non-zero data elements, signified by a ‘1’. The second bitmap
will concern only those blocks containing non-zero data elements from the first bitmap. In this bitmap a ‘0’ bit will represent a zero data element and a ‘1’ will represent a non-zero data element. The non-zero data elements still need to be encoded, but essentially all of the zero data values have now been encoded in the bitmaps. In the following chapters where examples of Block coding are shown, Huffman coding will be used to encode the non-zero data elements of each block.

There are two items of interest regarding this Block Coding technique. First, a high concentration of zeros in the transformed data is necessary in order for this technique to provide greater compression results than just a straight Huffman encoding of all the data values. Meaning that either the original image has to be highly continuous or the transform has to have been very successful. Second, just because there is a high concentration of zeros in the transformed data, does not necessarily mean that they are often in the appropriate blocks. This technique works best if there are a significant amount of blocks that are all zeros and the block size is also of a reasonable size.
CHAPTER 4

SUB SAMPLING

Transformation

The Sub Sampling transform can be described in two parts. First a small sample of the original image is kept that will be used to estimate the rest. Then the difference between the estimated data and the original is stored. This difference turns out to be highly compressible as described previously. A more detailed description of the transformation is described in the following steps, with the help of the grids in Figure 5 and Figure 6, representing pixels in the original image and the sub sample transform.

<table>
<thead>
<tr>
<th>X00</th>
<th>X01</th>
<th>X02</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>X10</td>
<td>X11</td>
<td>X12</td>
<td>...</td>
</tr>
<tr>
<td>X20</td>
<td>X21</td>
<td>X22</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 5. Original image pixel data

<table>
<thead>
<tr>
<th>A00 = X00</th>
<th>B00 = X01 - (X00+X02)/2</th>
<th>A01 = X02</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>C00 = X10 - (X00+X20)/2</td>
<td>D00 = X11 - (X01+X21+X10+X12)/2</td>
<td>C01 = X12 - (X02+X22)/2</td>
<td>...</td>
</tr>
<tr>
<td>A10 = X20</td>
<td>B10 = X21 - (X20+X22)/2</td>
<td>A11 = X22</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 6. Sub Sample transform data
1. The A's represent the sub sample of the whole image so they are left unchanged.

   For example: \( A_{00} = X_{00}, A_{01} = X_{02}, A_{02} = X_{04}, \) etc.

2. The D's are calculated from the 4 original pixels in the B & C positions.

   For example:
   
   \[
   D_{00} = X_{11} - \frac{(X_{10} + X_{12} + X_{01} + X_{21})}{4}
   \]
   
   \[
   D_{01} = X_{13} - \frac{(X_{12} + X_{14} + X_{03} + X_{23})}{4}
   \]

3. The B's and C's are calculated directly from the two nearest original pixels in the A positions.

   For example:
   
   \[
   B_{00} = X_{01} - \frac{(X_{00} + X_{02})}{2}
   \]
   
   \[
   B_{01} = X_{02} - \frac{(X_{02} + X_{04})}{2}
   \]
   
   \[
   C_{00} = X_{10} - \frac{(X_{00} + X_{20})}{2}
   \]
   
   \[
   C_{01} = X_{30} - \frac{(X_{20} + X_{40})}{2}
   \]

   The inverse transform is just the reverse of the above except that the original pixel values in the B and C positions need to be restored first, in order for them to be used to restore those in the D positions.

1. The A's represent the sub sample of the whole image so they are left unchanged.

   For example: \( X_{00} = A_{00}, X_{02} = A_{01}, X_{04} = A_{02}, \) etc.

2. The original pixel values in the B and C positions are restored directly from the two nearest original pixels in the A positions.

   For example:
   
   \[
   X_{01} = B_{00} + \frac{(X_{00} + X_{02})}{2}
   \]
\[ X_{02} = B_{01} + \frac{(X_{02} + X_{04})}{2} \]
\[ X_{10} = C_{00} + \frac{(X_{00} + X_{20})}{2} \]
\[ X_{30} = C_{01} + \frac{(X_{20} + X_{40})}{2} \]

3. The original pixel values in the D positions are restored from the 4 original pixels in the B & C positions.

For example:
\[ X_{11} = D_{00} + \frac{(X_{10} + X_{12} + X_{01} + X_{21})}{4} \]
\[ X_{13} = D_{01} + \frac{(X_{12} + X_{14} + X_{03} + X_{23})}{4} \]

Example frequency information describing the effect of the 1\textsuperscript{st} Level Sub Sample transform applied to the sample file “frame1.tif” is provided in Table 5 and Figure 7.
<table>
<thead>
<tr>
<th>Value Range</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>18412</td>
<td>15679</td>
<td>18198</td>
<td>52289</td>
</tr>
<tr>
<td>-1, 1</td>
<td>0</td>
<td>15922</td>
<td>14949</td>
<td>16385</td>
<td>47256</td>
</tr>
<tr>
<td>-2, 2</td>
<td>0</td>
<td>7628</td>
<td>9553</td>
<td>8539</td>
<td>25720</td>
</tr>
<tr>
<td>-4, 4</td>
<td>19</td>
<td>8913</td>
<td>10207</td>
<td>9465</td>
<td>28604</td>
</tr>
<tr>
<td>-8, 8</td>
<td>7205</td>
<td>7670</td>
<td>8541</td>
<td>7174</td>
<td>30590</td>
</tr>
<tr>
<td>-16, 16</td>
<td>10445</td>
<td>2440</td>
<td>2185</td>
<td>1582</td>
<td>16652</td>
</tr>
<tr>
<td>-32, 32</td>
<td>4657</td>
<td>236</td>
<td>174</td>
<td>55</td>
<td>5122</td>
</tr>
<tr>
<td>-64, 64</td>
<td>6869</td>
<td>217</td>
<td>122</td>
<td>33</td>
<td>7241</td>
</tr>
<tr>
<td>-128, 128</td>
<td>17938</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>17967</td>
</tr>
<tr>
<td>-256, 256</td>
<td>14307</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>14319</td>
</tr>
</tbody>
</table>

Table 5

1st Level Sub Sampling Frequencies (frame1.tif)
The Table 4.1 shows that the data for B, C, and D is now in a compressible form. The flaw with this transform is that the A’s remain in their original form and are not very compressible. In order to get better results the transform should be applied again, to just the A data set providing a 2nd level of transformation. After 4 levels of transformation over 99.5 percent of the original data has been transformed. This should produce satisfactory data for encoding. Sample frequencies after applying the Sub Sample transform to 4 levels of data is shown in Table 6 and Figure 8.
Table 6

4th Level Sub Sampling Frequencies (frame1.tif)

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>66401</td>
</tr>
<tr>
<td>-1, 1</td>
<td>59907</td>
</tr>
<tr>
<td>-2, 2</td>
<td>33110</td>
</tr>
<tr>
<td>-4, 4</td>
<td>37027</td>
</tr>
<tr>
<td>-8, 8</td>
<td>33244</td>
</tr>
<tr>
<td>-16, 16</td>
<td>12524</td>
</tr>
<tr>
<td>-32, 32</td>
<td>1922</td>
</tr>
<tr>
<td>-64, 64</td>
<td>909</td>
</tr>
<tr>
<td>-128, 128</td>
<td>465</td>
</tr>
<tr>
<td>-256, 256</td>
<td>251</td>
</tr>
</tbody>
</table>
Figure 8. 4\textsuperscript{th} Level Sub Sampling Frequencies (frame1.tif)

Encoding

Tables 7-10 and Figure 9 show compression ratios achieved by combining 4 levels of the Sub Sampling transform with the encoding techniques described previously.
### Table 7

**4th Level Sub Sampling + Huffman Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>120157</td>
<td>2.046</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>120371</td>
<td>2.043</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>120456</td>
<td>2.041</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>120520</td>
<td>2.040</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>119808</td>
<td>2.052</td>
</tr>
</tbody>
</table>

### Table 8

**4th Level Sub Sampling + LZ77 Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>167448</td>
<td>1.468</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>167714</td>
<td>1.466</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>167765</td>
<td>1.466</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>167895</td>
<td>1.465</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>166291</td>
<td>1.479</td>
</tr>
</tbody>
</table>
### Table 9

#### 4th Level Sub Sampling + LZW Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>138815</td>
<td>1.771</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>140182</td>
<td>1.754</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>140126</td>
<td>1.755</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>140108</td>
<td>1.755</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>138815</td>
<td>1.771</td>
</tr>
</tbody>
</table>

### Table 10

#### 4th Level Sub Sampling + (3x3)Block Coding Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>114731</td>
<td>2.143</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>115026</td>
<td>2.138</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>115209</td>
<td>2.134</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>115121</td>
<td>2.136</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>113947</td>
<td>2.158</td>
</tr>
</tbody>
</table>
Table 11

4th Level Sub Sampling + (4x4)Block Coding Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>114293</td>
<td>2.151</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>114597</td>
<td>2.146</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>114785</td>
<td>2.142</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>114695</td>
<td>2.144</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>113501</td>
<td>2.166</td>
</tr>
</tbody>
</table>

Figure 9. 4th Level Sub Sampling Compression Ratios
CHAPTER 5

DIFFERENTIAL PULSE CODE MODULATION (DPCM)

Transformation

The goal of DPCM is to keep one baseline pixel value and then go through the rest of the image estimating and saving the difference of the pixels next to each other, one at a time similar to a chain reaction. A simple example of DPCM can be performed on each scan line. The baseline pixel is the one in the first column. The transformed data for the rest of the line is the actual pixel value minus the value of the pixel immediately to the left. As in the Sub Sampling transform the original value minus the estimated value is kept. In this case the pixel to the left is the estimated pixel value. A description of this single pixel estimator DPCM is described in the following steps, with the help of the grids in Figure 10 and 11 representing the original and transformed data.

\[
\begin{array}{cccccc}
X_0 & X_1 & X_2 & X_3 & X_4 & \ldots \\
\end{array}
\]

Figure 10. Original image data

\[
\begin{array}{cccccc}
D_0 & D_1 & D_2 & D_3 & D_4 & \ldots \\
\end{array}
\]

Figure 11. DPCM transform data

1. \(D_0 = X_0\)
2. For \( j > 0 \): \( D_j = X_j - X_{j-1} \)

The single pixel estimator produces amiable results, but it can be improved upon. In the Sub Sampling transform two and sometimes four adjacent pixel values were used to calculate the estimated value. A more accurate estimate can be achieved if more than one adjacent pixel is used to calculate the estimate. It is impossible to use more than one pixel to calculate the estimate for every value in the DPCM transform. However, the majority of the estimates can be calculated by using two adjacent pixels as described in the following steps. \( X \) is the original pixel value, and \( D \) is the DPCM transform value.

1. The top left corner pixel will be the only value unchanged throughout the image.

\[ D_{0,0} = X_{0,0} \]

2. Perform a single scan line DPCM as described previously for the first row and column of the image.

First row, for \( j > 0 \): \( D_{0,j} = X_{0,j} - X_{0,j-1} \)

First column, for \( i > 0 \): \( D_{i,0} = X_{i,0} - X_{i-1,0} \)

3. For the second and subsequent rows perform a DPCM for each value after the first column using the original pixel immediately to the left and above to calculate your estimate.

For example:

\[ D_{i,j} = X_{i,j} - (X_{i-1,j} + X_{i,j-1})/2 \]

The inverse transform can be performed by simply adding the DPCM value to the same estimate. For example:

First row, for \( j > 0 \): \( X_{0,j} = D_{0,j} + X_{0,j-1} \)
First column, for \( i > 0 \): \( X_i,\theta = D_i,\theta + X_{i-1,\theta} \)

Remaining: \( X_i,j = D_{i,j} + (X_{i-1,j} + X_{i,j-1})/2 \)

No diagonal pixels were used for the calculating the estimated value. They were excluded simply because they are located far enough away from the pixel being estimated that they do not make a positive contribution towards the estimate. A good rule of thumb is that the best estimate always comes from the closest values.

Example frequency information describing the effect of the DPCM transform applied to the sample file “frame1.tif” is provided in Table 12 and Figure 12.
Table 12

DPCM Frequencies (frame1.tif)

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>65814</td>
</tr>
<tr>
<td>-1, 1</td>
<td>59608</td>
</tr>
<tr>
<td>-2, 2</td>
<td>33370</td>
</tr>
<tr>
<td>-4, 4</td>
<td>36840</td>
</tr>
<tr>
<td>-8, 8</td>
<td>34724</td>
</tr>
<tr>
<td>-16, 16</td>
<td>13415</td>
</tr>
<tr>
<td>-32, 32</td>
<td>1094</td>
</tr>
<tr>
<td>-64, 64</td>
<td>654</td>
</tr>
<tr>
<td>-128, 128</td>
<td>236</td>
</tr>
<tr>
<td>-256, 256</td>
<td>5</td>
</tr>
</tbody>
</table>
Tables 13-17 and Figure 13 show compression ratios achieved by combining the DPCM transform with the encoding techniques described previously.
Table 13

**DPCM + Huffman Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>118937</td>
<td>2.067</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>119182</td>
<td>2.063</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>119154</td>
<td>2.064</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>119334</td>
<td>2.061</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>118590</td>
<td>2.073</td>
</tr>
</tbody>
</table>

Table 14

**DPCM + LZ77 Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>166677</td>
<td>1.475</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>167034</td>
<td>1.472</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>166968</td>
<td>1.473</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>166996</td>
<td>1.472</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>165293</td>
<td>1.488</td>
</tr>
</tbody>
</table>
Table 15

**DPCM + LZW Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>137170</td>
<td>1.793</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>137605</td>
<td>1.787</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>137754</td>
<td>1.785</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>137931</td>
<td>1.783</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>136312</td>
<td>1.804</td>
</tr>
</tbody>
</table>

Table 16

**DPCM + (3x3)Block Coding Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>112903</td>
<td>2.178</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>113283</td>
<td>2.171</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>113387</td>
<td>2.169</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>113453</td>
<td>2.167</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>112154</td>
<td>2.192</td>
</tr>
</tbody>
</table>
Table 17

**DPCM + (4x4)Block Coding Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>112593</td>
<td>2.184</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>112954</td>
<td>2.177</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>113052</td>
<td>2.175</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>113125</td>
<td>2.174</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>111857</td>
<td>2.198</td>
</tr>
</tbody>
</table>

Figure 13. DPCM Compression Ratios
CHAPTER 6

INTEGER WAVELET COMPRESSION

Transformation

The Integer Wavelet transform is performed by applying a low-pass and a high-pass filter repeatedly to the original image data. The original image is denoted as \( X \), the low-pass component as \( Y \), and the high-pass component as \( Z \). When the transform is complete, the high-pass components will be in a form ready to encode, but that will be discussed later. What is involved with the transform will be described first. The grids in Figure 14-16 will be used to represent the original image and the resultant transform.

\[
\begin{array}{cccc}
X_{00} & X_{01} & X_{02} & X_{03} \\
X_{10} & X_{11} & X_{12} & X_{13} \\
X_{20} & X_{21} & X_{22} & X_{23} \\
X_{30} & X_{31} & X_{32} & X_{33} \\
\end{array}
\]

**Figure 14.** Original image pixel data

\[
\begin{array}{c|c}
Y_{00} = X_{00} + X_{01} & Y_{01} = X_{02} + X_{03} \\
Y_{10} = X_{10} + X_{11} & Y_{11} = X_{12} + X_{13} \\
Y_{20} = X_{20} + X_{21} & Y_{21} = X_{22} + X_{23} \\
Y_{30} = X_{30} + X_{31} & Y_{31} = X_{32} + X_{33} \\
\end{array}
\]

**Figure 15.** Integer Wavelet transform low-pass component data
\[
\begin{array}{ccc}
Z00 = X00 - X01 & Z01 = X02 - X03 \\
Z10 = X10 - X11 & Z11 = X12 - X13 \\
Z20 = X20 - X21 & Z21 = X22 - X23 \\
Z30 = X30 - X31 & Z31 = X32 - X33 \\
\ldots & \ldots & \ldots \\
\end{array}
\]

**Figure 16.** Integer Wavelet transform high-pass component data

Notice there are the same number of elements in the sum of the Y & Z data sets as there is in the original image. Actually this will only be the case if the width of the original image is even. When the width is odd, the last value of X should be duplicated so the calculation can be completed for that row. The inverse transform can be applied to the Y and Z data sets to restore the original X values as described in the grid in Figure 17.

\[
\begin{array}{ccc}
X00 = (Y00 + Z00) / 2 & X01 = (Y00 - Z00) / 2 \\
X10 = (Y10 + Z10) / 2 & X11 = (Y10 - Z10) / 2 \\
X20 = (Y20 + Z20) / 2 & X21 = (Y20 - Z20) / 2 \\
X30 = (Y30 + Z30) / 2 & X31 = (Y30 - Z30) / 2 \\
\ldots & \ldots & \ldots \\
\end{array}
\]

**Figure 17.** Integer Wavelet inverse transform data

The transform described so far is only the horizontal half of the first level of transformation. To complete the first level of this transform the same process is repeated, on the Y’s and Z’s except it is applied vertically instead of horizontally. The notation used above will be expanded to include the two new components of the transform. First we will look at the low-pass data set Y and transform it to the new components YY(low-pass) and YZ(high-pass) as shown in the grids in Figures 18 and 19.
$\begin{align*}
YY00 &= Y00 + Y10 \\
YY10 &= Y20 + Y30 \\
\cdots
\end{align*}$

**Figure 18.** Integer Wavelet vertical low-pass transform data applied to the Y data set

$\begin{align*}
YY01 &= Y01 + Y11 \\
YY11 &= Y21 + Y21 \\
\cdots
\end{align*}$

$\begin{align*}
YZ00 &= Y00 - Y10 \\
YZ10 &= Y20 - Y30 \\
\cdots
\end{align*}$

**Figure 19.** Integer Wavelet vertical high-pass transform data applied to the Y data set

$\begin{align*}
YZ01 &= Y01 - Y11 \\
YZ11 &= Y21 - Y21 \\
\cdots
\end{align*}$

Similarly, the transform is applied to the Z data set is shown in the grids in Figures 20 and 21.

$\begin{align*}
ZY00 &= Z00 + Z10 \\
ZY10 &= Z20 + Z30 \\
\cdots
\end{align*}$

**Figure 20.** Integer Wavelet vertical low-pass transform data applied to the Z data set

$\begin{align*}
ZY01 &= Z01 + Z11 \\
ZY11 &= Z21 + Z21 \\
\cdots
\end{align*}$

$\begin{align*}
ZZ00 &= Z00 - Z10 \\
ZZ10 &= Z20 - Z30 \\
\cdots
\end{align*}$

**Figure 21.** Integer Wavelet vertical high-pass transform data applied to the Z data set

$\begin{align*}
ZZ01 &= Z01 - Z11 \\
ZZ11 &= Z21 - Z21 \\
\cdots
\end{align*}$

The inverse transform for these components is shown in the grids in Figures 22 and 23.
\[
\begin{array}{c|c}
Y00 = (YY00 + YZ00) / 2 & Y01 = (YY01 + YZ01) / 2 \\
Y10 = (YY00 - YZ00) / 2 & Y11 = (YY01 - YZ01) / 2 \\
\vdots & \vdots
\end{array}
\]

Figure 22. Integer Wavelet inverse transform to restore the Y data set

\[
\begin{array}{c|c}
Z00 = (ZY00 + ZZ00) / 2 & Z01 = (ZY01 + ZZ01) / 2 \\
Z10 = (ZY00 - ZZ00) / 2 & Z11 = (ZY01 - ZZ01) / 2 \\
\vdots & \vdots
\end{array}
\]

Figure 23. Integer Wavelet inverse transform to restore the Z data set

This completes one level of the transform. Sample frequency information for the YY, YZ, ZY, and ZZ when applied to the “frame 1.tif” file is shown in Table 18. A histogram of the total set of transformed data is shown in Figure 24.
Table 18

1st Level Integer Wavelet Frequencies (frame1.tif)

<table>
<thead>
<tr>
<th>Value Range</th>
<th>YY</th>
<th>YZ</th>
<th>ZY</th>
<th>ZZ</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11375</td>
<td>13298</td>
<td>13854</td>
<td>38527</td>
</tr>
<tr>
<td>-1, 1</td>
<td>0</td>
<td>8584</td>
<td>10902</td>
<td>11963</td>
<td>31449</td>
</tr>
<tr>
<td>-2, 2</td>
<td>0</td>
<td>7328</td>
<td>7149</td>
<td>8289</td>
<td>22766</td>
</tr>
<tr>
<td>-4, 4</td>
<td>0</td>
<td>10263</td>
<td>7688</td>
<td>9144</td>
<td>27095</td>
</tr>
<tr>
<td>-8, 8</td>
<td>0</td>
<td>10666</td>
<td>9302</td>
<td>10189</td>
<td>30157</td>
</tr>
<tr>
<td>-16, 16</td>
<td>0</td>
<td>9644</td>
<td>9008</td>
<td>6856</td>
<td>25508</td>
</tr>
<tr>
<td>-32, 32</td>
<td>5274</td>
<td>3081</td>
<td>3498</td>
<td>1049</td>
<td>12902</td>
</tr>
<tr>
<td>-64, 64</td>
<td>12192</td>
<td>247</td>
<td>351</td>
<td>56</td>
<td>12846</td>
</tr>
<tr>
<td>-128, 128</td>
<td>4658</td>
<td>110</td>
<td>142</td>
<td>34</td>
<td>4944</td>
</tr>
<tr>
<td>-256, 256</td>
<td>6997</td>
<td>119</td>
<td>102</td>
<td>6</td>
<td>7224</td>
</tr>
<tr>
<td>-512, 512</td>
<td>18043</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>18066</td>
</tr>
<tr>
<td>-1024, 1024</td>
<td>14276</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14276</td>
</tr>
</tbody>
</table>
Now the YZ, ZY, and ZZ data sets are in a form ready to encode. However, like the A component of the Sub Sample transform, the YY data set is not. For better results, it is necessary to repeat the procedure on the successive YY data sets completing 3 levels of the transform as show in Table 19 and Figure 25. Completing 4 levels of the transform results in a range of values that are outside the scope of the Huffman coding routine implemented earlier in this thesis, so only 3 levels of the transform can be used.
<table>
<thead>
<tr>
<th>Value Range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>46237</td>
</tr>
<tr>
<td>-1, 1</td>
<td>35087</td>
</tr>
<tr>
<td>-2, 2</td>
<td>26046</td>
</tr>
<tr>
<td>-4, 4</td>
<td>32814</td>
</tr>
<tr>
<td>-8, 8</td>
<td>37283</td>
</tr>
<tr>
<td>-16, 16</td>
<td>33287</td>
</tr>
<tr>
<td>-32, 32</td>
<td>16935</td>
</tr>
<tr>
<td>-64, 64</td>
<td>8179</td>
</tr>
<tr>
<td>-128, 128</td>
<td>3530</td>
</tr>
<tr>
<td>-256, 256</td>
<td>1570</td>
</tr>
<tr>
<td>-512, 512</td>
<td>931</td>
</tr>
<tr>
<td>-1024, 1024</td>
<td>954</td>
</tr>
<tr>
<td>-2048, 2048</td>
<td>355</td>
</tr>
<tr>
<td>-4096, 4096</td>
<td>460</td>
</tr>
<tr>
<td>-8192, 8192</td>
<td>1221</td>
</tr>
<tr>
<td>-16384, 16384</td>
<td>871</td>
</tr>
</tbody>
</table>
Encoding

This transform will produce data that will overflow unsigned character arithmetic to the point where the inverse transform cannot recover the original data. This means that the LZ family of encoding techniques that rely on unsigned characters for input symbols cannot be used in the encoding stage of this compression scheme. It may be possible to modify the LZ encoding techniques to handle a wider range of input symbols, but that is beyond the scope of this thesis. Huffman and Block coding techniques seem to achieve better compression ratios than the LZ family when used with the transforms described in this thesis anyway. Tables 20-22 and Figure 26 show compression ratios achieved by combining 3 levels of the Integer Wavelet transform with the Huffman and Block coding techniques described previously.
### Table 20

**3rd Level Integer Wavelet + Huffman Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>176605</td>
<td>1.392</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>176501</td>
<td>1.393</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>176559</td>
<td>1.393</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>176728</td>
<td>1.391</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>175399</td>
<td>1.402</td>
</tr>
</tbody>
</table>

### Table 21

**3rd Level Integer Wavelet + (3x3)Block Coding Compression Ratios**

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>168948</td>
<td>1.455</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>168914</td>
<td>1.456</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>169097</td>
<td>1.454</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>169141</td>
<td>1.454</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>167468</td>
<td>1.468</td>
</tr>
</tbody>
</table>
Table 22

3rd Level Integer Wavelet + (4x4) Block Coding Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>168521</td>
<td>1.459</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>168469</td>
<td>1.460</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>168640</td>
<td>1.458</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>168670</td>
<td>1.458</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>166957</td>
<td>1.473</td>
</tr>
</tbody>
</table>

Figure 26. 3rd Level Integer Wavelet Compression Ratios
CHAPTER 7

FLOATING POINT WAVELETS, AN ALTERNATIVE TO LOSSLESS IMAGE COMPRESSION

Chapter 3 stated that in order for a compression technique to be lossless the transformation must have an inverse that will bring the data back to its exact original form. This chapter describes the use of a floating point Wavelet transform and its inverse that, due to rounding/truncation error cannot restore the data to its exact original form. The error it commits is very small and depending on the image, may not be noticeable to the human eye. It will be shown that the use of this transform will produce data that has many more zeros than those discussed previously and as a result higher compression ratios as well.

Transformation

The method of transformation is very similar to the Integer Wavelet described in Chapter 6. There is a low-pass and high-pass component of the transform that is applied both horizontally and vertically to the original image data. The resulting data has four parts, only one of which is purely a low-pass data set. To provide better results it should be performed repeatedly, as in the Sub Sample and Integer Wavelet transforms described previously. To simplify the process description, the notation of grids and data sets used in
Chapter 6 will also be used here. The horizontal part of the transform can be described by
applying the following filters to each scan line of the image.

Constant Multipliers: \( a_0 = 0.859312, a_1 = 0.353553, a_2 = -0.076103 \)

Low-Pass Filter: \( a_2 + a_1 + a_0 + a_1 + a_2 \)

High-Pass Filter: \( a_2 - a_1 + a_0 - a_1 + a_2 \)

1. Apply the low-pass filter, centered about the \( a_0 \), to every even pixel in each scan line
to calculate the \( Y \) values.

2. Apply the high-pass filter, centered about the \( a_0 \), to every odd pixel in each scan line
to calculate the \( Z \) values.

NOTE: At the edges of the image, use values on the inside of the image reflected
to those that would lie outside. For example to calculate the first value of \( Y \):

\[
Y_{00} = X_{02}a_2 + X_{01}a_1 + X_{00}a_0 + X_{01}a_1 + X_{02}a_2
\]

A more formal description of the horizontal application of the transform is shown
in the grids in Figures 27-29.

<table>
<thead>
<tr>
<th>X₀₀</th>
<th>X₀₁</th>
<th>X₀₂</th>
<th>X₀₃</th>
<th>X₀₄</th>
<th>X₀₅</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁₀</td>
<td>X₁₁</td>
<td>X₁₂</td>
<td>X₁₃</td>
<td>X₁₄</td>
<td>X₁₅</td>
<td>...</td>
</tr>
</tbody>
</table>

... ... ... ... ... ...

*Figure 27. Original image pixel data*
\[
\begin{align*}
Y_{00} &= X_{02}a_2 + X_{01}a_1 + X_{00}a_0 + \\
&\quad X_{01}a_1 + X_{02}a_2 \\
Y_{10} &= X_{12}a_2 + X_{11}a_1 + X_{10}a_0 + \\
&\quad X_{11}a_1 + X_{12}a_2 \\
Y_{01} &= X_{00}a_2 + X_{01}a_1 + X_{02}a_0 + \\
&\quad X_{03}a_1 + X_{04}a_2 \\
Y_{11} &= X_{10}a_2 + X_{11}a_1 + X_{12}a_0 + \\
&\quad X_{13}a_1 + X_{14}a_2 \\
\end{align*}
\]

Figure 28. Floating Point Wavelet transform low-pass component data

\[
\begin{align*}
Z_{00} &= X_{03}a_2 - X_{00}a_1 + X_{01}a_0 - \\
&\quad X_{02}a_1 + X_{02}a_2 \\
Z_{10} &= X_{13}a_2 - X_{10}a_1 + X_{11}a_0 - \\
&\quad X_{12}a_1 + X_{13}a_2 \\
Z_{01} &= X_{01}a_2 - X_{02}a_1 + X_{03}a_0 - \\
&\quad X_{04}a_1 + X_{05}a_2 \\
Z_{11} &= X_{11}a_2 - X_{12}a_1 + X_{13}a_0 - \\
&\quad X_{14}a_1 + X_{15}a_2 \\
\end{align*}
\]

Figure 29. Floating Point Wavelet transform high-pass component data

The same process is repeated along the columns of the Y and Z data sets, producing four new data sets: YY, YZ, ZY, and ZZ as in Chapter 6.

The inverse of this transform is calculated by applying the opposite high/low filters to the corresponding Y and Z data elements, depending on if it is an even or odd pixel being calculated. As before with the X values, use reflected values of Y and Z in the calculation. This inverse transform is shown in the grid in Figure 30.

\[
\begin{align*}
X_{00} &= Y_{01}a_2 - Z_{00}a_1 + Y_{00}a_0 - \\
&\quad Z_{00}a_1 + Y_{01}a_2 \\
X_{10} &= Y_{11}a_2 - Z_{10}a_1 + Y_{10}a_0 - \\
&\quad Z_{10}a_1 + Y_{11}a_2 \\
X_{01} &= Z_{01}a_2 + Y_{00}a_1 + Z_{00}a_0 + \\
&\quad Y_{01}a_1 + Z_{01}a_2 \\
X_{11} &= Z_{11}a_2 + Y_{10}a_1 + Z_{10}a_0 + \\
&\quad Y_{11}a_1 + Z_{11}a_2 \\
\end{align*}
\]

Figure 30. Floating Point Wavelet inverse transform data

A similar inverse process can be used to restore the Y and Z from the YY, YZ, ZY, and ZZ data sets.
This completes the description of one level of the transform. Sample frequency range information for the YY, YZ, ZY, and ZZ values when applied to the “frame1.tif” file is shown in Table 23. A histogram of the total set of transformed data is shown in Figure 31.

Table 23

<table>
<thead>
<tr>
<th>Value Range</th>
<th>YY</th>
<th>YZ</th>
<th>ZY</th>
<th>ZZ</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>23507</td>
<td>32438</td>
<td>34898</td>
<td>90845</td>
</tr>
<tr>
<td>-1, 1</td>
<td>4</td>
<td>12146</td>
<td>7837</td>
<td>9672</td>
<td>29659</td>
</tr>
<tr>
<td>-2, 2</td>
<td>12</td>
<td>7224</td>
<td>5523</td>
<td>6237</td>
<td>18996</td>
</tr>
<tr>
<td>-4, 4</td>
<td>66</td>
<td>8810</td>
<td>7267</td>
<td>6650</td>
<td>22793</td>
</tr>
<tr>
<td>-8, 8</td>
<td>99</td>
<td>7412</td>
<td>5987</td>
<td>3506</td>
<td>17004</td>
</tr>
<tr>
<td>-16, 16</td>
<td>9026</td>
<td>1932</td>
<td>1842</td>
<td>402</td>
<td>13202</td>
</tr>
<tr>
<td>-32, 32</td>
<td>8279</td>
<td>221</td>
<td>207</td>
<td>41</td>
<td>8748</td>
</tr>
<tr>
<td>-64, 64</td>
<td>4810</td>
<td>119</td>
<td>175</td>
<td>29</td>
<td>5133</td>
</tr>
<tr>
<td>-128, 128</td>
<td>6903</td>
<td>33</td>
<td>164</td>
<td>4</td>
<td>7104</td>
</tr>
<tr>
<td>-256, 256</td>
<td>18038</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>18042</td>
</tr>
<tr>
<td>-512, 512</td>
<td>14198</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>14201</td>
</tr>
</tbody>
</table>
As in Chapter 6; YZ, ZY, and ZZ data sets are in a compressible form and YY is not. Repeat the process for each successive YY data sets until 4 levels of transformation have been completed to provide better data for the encoding stage of the compression process. Example frequency ranges after the 4 levels of transformation are shown in Table 24 and Figure 32.
Table 24

4\textsuperscript{th} Level Floating Point Wavelet Frequencies (frame1.tif)

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100542</td>
</tr>
<tr>
<td>-1, 1</td>
<td>45955</td>
</tr>
<tr>
<td>-2, 2</td>
<td>25808</td>
</tr>
<tr>
<td>-4, 4</td>
<td>30566</td>
</tr>
<tr>
<td>-8, 8</td>
<td>26303</td>
</tr>
<tr>
<td>-16, 16</td>
<td>11085</td>
</tr>
<tr>
<td>-32, 32</td>
<td>2537</td>
</tr>
<tr>
<td>-64, 64</td>
<td>1103</td>
</tr>
<tr>
<td>-128, 128</td>
<td>665</td>
</tr>
<tr>
<td>-256, 256</td>
<td>433</td>
</tr>
<tr>
<td>-512, 512</td>
<td>113</td>
</tr>
<tr>
<td>-1024, 1024</td>
<td>132</td>
</tr>
<tr>
<td>-2048, 2048</td>
<td>290</td>
</tr>
<tr>
<td>-4096, 4096</td>
<td>228</td>
</tr>
</tbody>
</table>
Tables 25-27 and Figure 33 show compression ratios achieved by combining 4 levels of the Floating Point Wavelet transform with the Huffman and Block coding techniques.

Figure 32. 4th Level Floating Point Wavelet Frequencies (frame1.tif)
Table 25

4th Level Floating Point Wavelet + Huffman Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>112747</td>
<td>2.181</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>112787</td>
<td>2.180</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>112884</td>
<td>2.178</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>112905</td>
<td>2.178</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>111842</td>
<td>2.199</td>
</tr>
</tbody>
</table>

Table 26

4th Level Floating Point Wavelet + (3x3)Block Coding Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>107029</td>
<td>2.297</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>107064</td>
<td>2.297</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>107340</td>
<td>2.291</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>107275</td>
<td>2.292</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>106016</td>
<td>2.319</td>
</tr>
</tbody>
</table>
Table 27

4th Level Floating Point Wavelet + (4x4) Block Coding Compression Ratios

<table>
<thead>
<tr>
<th>Sample File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>245894</td>
<td>106678</td>
<td>2.305</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>245894</td>
<td>106734</td>
<td>2.304</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>245894</td>
<td>106981</td>
<td>2.298</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>245894</td>
<td>106967</td>
<td>2.299</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>245894</td>
<td>105658</td>
<td>2.327</td>
</tr>
</tbody>
</table>

Figure 33. 4th Level Floating Point Wavelet Compression Ratios
CHAPTER 8

CONCLUSIONS

As mentioned previously, the purpose of this thesis was to achieve higher lossless image compression ratios by using a more sophisticated compression model than the various LZ encoding techniques used today. For comparison, a summary of the example compression ratios given in Chapter 2 are grouped together in Table 28 and Figure 34. The Huffman coding information is not included because, by itself, it is not considered a current lossless image compression technique. Huffman coding was introduced in Chapter 2 primarily because of its use in the encoding stage of the compression model introduced in this thesis. The PKZIP compression ratios are given as an example of the combined LZ77 and Huffman coding algorithm discussed in Chapter 2. Because PKZIP is a frequently used commercial product, using its compression ratios for comparison is important to the conclusions of this thesis. The compression ratios for each transform in Chapters 4-7 when combined with the 4x4 Block coding scheme are also included in Table 28 and Figure 34. The 4x4 Block coding scheme was chosen because it consistently produced the highest compression ratios for the examples in Chapters 4-7.
Table 28

Sample compression ratios summary

<table>
<thead>
<tr>
<th>Sample File</th>
<th>LZ77</th>
<th>PKZIP</th>
<th>LZW</th>
<th>SS + 4x4B</th>
<th>DPCM + 4x4B</th>
<th>IW + 4x4B</th>
<th>FPW + 4x4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame1.tif</td>
<td>1.306</td>
<td>1.529</td>
<td>1.368</td>
<td>2.151</td>
<td>2.184</td>
<td>1.459</td>
<td>2.305</td>
</tr>
<tr>
<td>frame2.tif</td>
<td>1.303</td>
<td>1.526</td>
<td>1.364</td>
<td>2.146</td>
<td>2.177</td>
<td>1.460</td>
<td>2.304</td>
</tr>
<tr>
<td>frame3.tif</td>
<td>1.303</td>
<td>1.527</td>
<td>1.369</td>
<td>2.142</td>
<td>2.175</td>
<td>1.458</td>
<td>2.298</td>
</tr>
<tr>
<td>frame4.tif</td>
<td>1.303</td>
<td>1.525</td>
<td>1.349</td>
<td>2.144</td>
<td>2.174</td>
<td>1.458</td>
<td>2.299</td>
</tr>
<tr>
<td>frame5.tif</td>
<td>1.319</td>
<td>1.546</td>
<td>1.364</td>
<td>2.166</td>
<td>2.198</td>
<td>1.473</td>
<td>2.327</td>
</tr>
</tbody>
</table>

Figure 34. Sample compression ratios summary

With exception of the Integer Wavelet transform, the data in Table 8.1 and Graph 8.1 shows that the compression model introduced in this thesis can consistently produce compression ratios higher than the various LZ encoding techniques used today.
Future Research Suggestions

There is always going to be an interest in compression research, there is much work still to be done in this area. This thesis certainly did not cover the spectrum of compression algorithms. Even though lossless image compression is a smaller field than lossy image compression there is still a good deal of information available that was not included in this document.

Transformation

From the different compression ratios achieved by using different transformations in this thesis, it would seem that transformation is the key to image compression. Many transforms may yet be discovered that could contribute to the field of image compression.

- Only one example of an integer wavelet was described; additional and more effective transforms could be found and implemented.
- A transform specifically targeted to produce repetitious strings could be researched that might improve LZ encoding for images.

Encoding

There are other encoding techniques that were not included in this thesis. For example:

- The added Huffman coding to the LZ77 compression algorithm mentioned in Chapter 2 was not implemented. The commercial product PKZIP was used as an example of this technique instead. Because it was not implemented it could not be used as the second stage of the encoding model for any of the transforms.
• Arithmetic coding, a minimum redundancy coding technique similar to Huffman, may provide better results than the Huffman coding used here.

• More complex run length coding algorithms might be applicable as well. The Block coding algorithm was only used to capture the zero run-length. It is possible that it could be used to capture the -1, 1 or maybe -2, 2 ranges in similar bitmap coding schemes.

• The possibility of enhancing the LZ family of encoders to accept input of more than just unsigned characters could be explored. LZ encoders, as they are, do not perform as well as Huffman coding for the transforms described here.

• Floating point wavelet transforms produce the most compressible data. It is conceivable that an encoding scheme could be devised to keep track of the errors committed by these transforms so that the image can be completely restored.

A great deal was learned in this study of lossless image compression. A paramount discovery was that one cannot always tell if an idea will work until one tries it. Sometimes the simplest techniques work the best. The DPCM, for example, was the most simple of all the transforms presented and worked as well or better than any of the other techniques.
APPENDIX I

SOURCE CODE

All source code was written in using Borland C++ 5.01. Windows and other support routines were not included.

Huffman Encoding

/*

huff15.h
*/

ifndef __HUFF15_H
#define __HUFF15_H

#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>

#define MAX 32767
#define NUMBITS 15

bool huff15(FILE *fp, unsigned int huge *X, long numX);
bool unhuff15(FILE *fp, unsigned int huge *X);

#endif

/*

huff15.cpp - 15-bit Huffman coding
handles input symbols upto 15-bit unsigned int
*/

#include "huff15.h"
void slownormal(void);
unsigned int skip(unsigned int *x, unsigned int *ndx, unsigned int num);
void normalize(unsigned int *X, unsigned int *ndx, unsigned int num);
void normalize2(unsigned int *X, unsigned int *ndx, unsigned int start, unsigned int end);
void Hqs(unsigned int *item, int left, int right, unsigned int *ndx);
unsigned int buildtree(void);
void shiftdown(unsigned int num);
bool outputtree(FILE *fp, unsigned int);
void traverse(FILE *fp, unsigned int node);
bool outputsymbols(FILE *fp, unsigned int huge *X, long numX, unsigned int root);

static int outpos; // global variable for output byte position
static unsigned char outbyte; // global variable for output byte

static unsigned int *symfreq; // global symbol frequency table
static unsigned int *symindx; // global index to symbol frequency table
static unsigned int *parfreq; // global parent frequency table
static unsigned int *parindx; // global index to parent frequency table
static unsigned int *child0; // global zero child table
static unsigned int *child1; // global one child table

//static FILE *test;

bool huffl5(FILE *fp, unsigned int huge *X, long numX)
{
    long numcodes;
    unsigned int huge *Xptr, root;
    int i;

    // Testing
    // test = fopen("test.dat","w");
    // fprintf(test,"Symbols: (numX=%ld)\n",numX);
    /*
    numcodes = numX;
    Xptr = X;
    while(numcodes-- > 0) fprintf(test," %u\n",*Xptr++);
    fprintf(test,\"\n");
    */

    // write numX to first 4 bytes of file and initialize output byte & pos
    numcodes = numX;
    for(i=3; i>=0; i--) {
        numcodes = (numX >> (8*i));
    }
outbyte = (unsigned char)(numcodes % 256);
putc(outbyte,fp);

// fprintf(test,"%d\n",outbyte);
}
outbyte = 0;
outbpos = 0;
symfreq = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(symfreq==NULL) return false;
symindx = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(symindx==NULL) {
    free(symfreq);
    return false;
}
parfreq = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(parfreq==NULL) {
    free(symfreq);
    free(symindx);
    return false;
}
parindx = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(parindx==NULL) {
    free(symfreq);
    free(symindx);
    free(parfreq);
    return false;
}
child0 = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(child0==NULL) {
    free(symfreq);
    free(symindx);
    free(parfreq);
    free(parindx);
    return false;
}
child1 = (unsigned int *) calloc(MAX,sizeof(unsigned int));
if(child1==NULL) {
    free(symfreq);
    free(symindx);
    free(parfreq);
    free(parindx);
    free(child0);
    return false;
}
// initialize tables
for(i=0; i<MAX; i++){
    symfreq[i] = 0;
    symindx[i] = i;
    parfreq[i] = 0;
    parindx[i] = i;
    child0[i] = MAX;
    child1[i] = MAX;
}

// calculate the frequency of each symbol
numcodes = numX;
Xptr = X;
while(numcodes-- > 0) {
    if(*Xptr >= MAX) {  // return false if number out of bounds
        free(symfreq);
        free(symindx);
        free(parfreq);
        free(parindx);
        free(child0);
        free(child1);
        return false;
    }
    symfreq[*Xptr]++;
    // normalize if necessary
    if(symfreq[*Xptr] >= (unsigned int)MAX*2) slownormal();
    Xptr++;
}
Hqs(symfreq, 0, MAX-1, symindx);

/* Testing
fprintti[0,"\nFrequencies:\n");
for(root=0; root<MAX; root++) {
    if(symfreq[symindx[root]]>0) {
        fprintf(test, " %u - %u\n",symindx[root],symfreq[symindx[root]]);
    }
}
fprintf(test,"\n");
*/

if((root=buildtree() == 0) {
    free(symfreq);
    free(symindx);
    free(parfreq);
free(parindx);
free(child0);
free(child1);
return false;
}

// now that tree is built, these aren't necessary
free(symfreq);
free(parfreq);
// keep symindx to immediately access each symbol

// fprintf(test,"Huffman Tree Info:"
;if(!outputtree(fp, root)) {
    free(symindx);
    free(child0);
    free(child1);
    return false;
}
// fprintf(test,""
;

// fprintf(test,"Encoding Symbols:"
;if(!outputsymbols(fp, X, numX, root)) {
    free(symindx);
    free(parindx);
    free(child0);
    free(child1);
    return false;
}

free(symindx);
free(parindx);
free(child0);
free(child1);

// fclose(test);
return true;
}

unsigned int skip(unsigned int *x, unsigned int *ndx, unsigned int num)
{
    unsigned int low, high, mid;

    low = 0;
    high = num;
while(low<high){
    mid = (low+high)/2;
    if(x[ndx[mid]]==0) low=mid+1;
    else high=mid;
}
return high;

void slownormal(void)
{
    unsigned int i;
    for(i=0; i<MAX; i++)
    if(symfreq[i]>0) symfreq[i] = symfreq[i]/2 + 1;
}

void normalize(unsigned int *X, unsigned int *ndx, unsigned int num)
{
    unsigned int i;
    for(i=skip(X,ndx,num); i<num; i++) X[ndx[i]] = X[ndx[i]]/2 + 1;
}

void normalize2(unsigned int *X, unsigned int *ndx, unsigned int start, unsigned int end)
{
    unsigned int i;
    for(i=start; i<end; i++) X[ndx[i]] = X[ndx[i]]/2 + 1;
}

void Hqs(unsigned int *item, int left, int right, unsigned int *ndx)
{
    int i,j;
    unsigned int x,y;

    i=left; j=right;
    x=ndx[(left+right)/2];

do {
    while(item[ndx[i]]<item[x] && i<right) i++;
    while(item[x]<item[ndx[j]] && j>left) j--;
    if(i<=j) {
        if( (i==j) && (ndx[i]==ndx[j]) ){
            /* swap ndx[i] & ndx[j] */
    
}}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
unsigned int buildtree(void)
{
    unsigned int nextparent, nextsymbol, first, x1, x2, y1, y2;

    // initialize nextparent and nextsymbol
    nextparent = MAX;
    for(nextsymbol=0; nextsymbol<MAX; nextsymbol++) // skip past 0 freq symbols
        if(symfreq[symindx[nextsymbol]] > 0) break;

    // first parent in tree is connected to two smallest freq symbols
    x1 = symfreq[symindx[nextsymbol]];
    x2 = symfreq[symindx[nextsymbol+1]];
    while(((long)x1 + (long)x2) >= (long)MAX*2) {
        normalize2(symfreq,symindx,nextsymbol,MAX);
        normalize(parfreq,parindx,nextparent-MAX);
        x1 = x1/2+1;
        x2 = x2/2+1;
    }
    parfreq[nextparent-MAX] = x1 + x2;
    child0[nextparent-MAX] = symindx[nextsymbol++];
    child1[nextparent-MAX] = symindx[nextsymbol++];
    nextparent++;
    // Not necessary to sort, only one item in list

    while(1) {

        // get freq of smallest 2 symbols
        if(nextsymbol < MAX) x1 = symfreq[symindx[nextsymbol]];
        else x1 = (unsigned int)MAX*2; // set x1 the max value so it won't be used
        if(nextsymbol+1 < MAX) x2 = symfreq[symindx[nextsymbol+1]];
        else x2 = (unsigned int)MAX*2; // set x2 the max value so it won't be used
// get freq of smallest 2 parents
first = skip(parfreq, parindx, nextparent-MAX);
if(first>=nextparent-MAX) return(0);
y1 = parfreq[parindx[first]]; // gotta be at least one
if(first+1 < nextparent-MAX) y2 = parfreq[parindx[first+1]];
else { // only one parent
  // if only one parent and all symbols have been added to the tree
  if(nextsymbol>=MAX) return(parindx[first]);
y2 = (unsigned int)MAX*2; // set y2 the max value so it won't be used
}

// find out which nodes are to be added to the tree
if(x2 <= y1) { // 2 symbols x1 & x2
  while(((long)x1 + (long)x2) >= (long)MAX*2) {
    normalize2(symfreq, symindx, nextsymbol, MAX);
    normalize(parfreq, parindx, nextparent-MAX);
    x1 = x1/2+1;
    x2 = x2/2+1;
  }
  parfreq[nextparent-MAX] = x1 + x2;
  child0[nextparent-MAX] = symindx[nextsymbol++];
  child1[nextparent-MAX] = symindx[nextsymbol++];
} else if(y2 < x1) { // 2 parents y1 & y2
  while(((long)y1 + (long)y2) >= (long)MAX*2) {
    normalize2(symfreq, symindx, nextsymbol, MAX);
    normalize(parfreq, parindx, nextparent-MAX);
    y1 = y1/2+1;
    y2 = y2/2+1;
  }
  parfreq[nextparent-MAX] = y1 + y2;
  child0[nextparent-MAX] = parindx[first]+MAX; //y1
  child1[nextparent-MAX] = parindx[first+1]+MAX; //y2
  // set frequencies of old parents to 0, so they won't be added again
  parfreq[parindx[first]] = 0;
  shiftdown(first);
  parfreq[parindx[first+1]] = 0;
  shiftdown(first+1);
} else if(x1 <= y1) { // x1 & y1
  while(((long)x1 + (long)y1) >= (long)MAX*2) {
normalize2(symfreq, symindx, nextsymbol, MAX);
normalize(parfreq, parindx, nextparent-MAX);
x1 = x1/2+1;
y1 = y1/2+1;
}
parfreq[nextparent-MAX] = x1 + y1;
child0[nextparent-MAX] = symindx[nextsymbol++];     //x1
child1[nextparent-MAX] = parindx[first]+MAX;    //y1
// set frequencies of old parents to 0, so they won't be added again
parfreq[parindx[first]] = 0;
shiftdown(first);
}
else {     // y1 & x1
while(((long)x1 + (long)y1) >= (long)MAX*2) {
    normalize2(symfreq, symindx, nextsymbol, MAX);
    normalize(parfreq, parindx, nextparent-MAX);
    x1 = x1/2+1;
y1 = y1/2+1;
}
parfreq[nextparent-MAX] = x1 + y1;
child0[nextparent-MAX] = parindx[first]+MAX;    //y1
child1[nextparent-MAX] = symindx[nextsymbol++];     //x1
// set frequencies of old parents to 0, so they won't be added again
parfreq[parindx[first]] = 0;
shiftdown(first);
}

// Increment next avail parent & Sort parent frequency indeces
shiftdown(nextparent-MAX);
nextparent++;
}

void shiftdown(unsigned int num)
{
    unsigned int i, x;
    for(i=num; i>0 && parfreq[parindx[i]]<parfreq[parindx[i-1]]; i--) {
        x = parindx[i];
        parindx[i] = parindx[i-1];
        parindx[i-1] = x;
    }
bool outputtree(FILE *fp, unsigned int root)
{
    traverse(fp, root);
    return true;
}

void traverse(FILE *fp, unsigned int node)
{
    int i;

    if(childO[node] >= MAX) {
        // to build the parent back tracking list
        parindx[childO[node]-MAX] = node;
        // fprintf(test,"0");
        outbpos++; // 0-bit
        if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
        traverse(fp,childO[node]-MAX);
    } else {
        // to build the parent back tracking list
        symindx[childO[node]] = node;
        // fprintf(test,"l - %u\n", cMldO[node]);
        outbyte |= (l«outbpos++); // 1-bit, signifying a symbol to follow
        if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
        // output symbol
        for(i=0; i<NUMBITS; i++) {
            if(childO[node] & (1<i)) outbyte |= (l«outbpos);
            outbpos++;
            if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
        }
    }
    if(child1[node] >= MAX) {
        // to build the parent back tracking list
        parindx[child1[node]-MAX] = node;
    }
}
// don't output anything, just go there
traverse(fp, childl[node]-MAX);
} else {
// to build the parent back tracking list
symindx[childl[node]] = node;

// fprintf(test, "1 - %u\n", childl[node]);
outbyte |= (1<<outbpos++); // 1-bit, signifying a symbol to follow
if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
// output symbol
for(i=0; i<NUMBITS; i++) {
  if(childl[node] & (1<<i)) outbyte |= (1<<outbpos);
  outbpos++;
  if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
}
}

bool outputsymbols(FILE *fp, unsigned int huge *X, long numX, unsigned int root) {
  unsigned int huge *Xptr, back, last;
  char buffer[1000];
  int i;

  Xptr = X;
  while(numX-- > 0) {
    // backtrack tree until reaching root, counting number of steps
    last = *Xptr;
    back = symindx[last];
    for(i=1; i<1000 && back!=root; i++) {
      back = parindx[back];
    }
    if(back!=root) return false;

    // now length of bit-string is known, put into a tmp buffer, in reverse order
    buffer[i--] = '0';
    last = *Xptr;
    back = symindx[last];
    while(back!=root) {
      if(child0[back] < MAX) {
        if(child0[back] == last) buffer[i--] = '0';
      } else {
        if(child0[back]-MAX == last) buffer[i--] = '0';
      }
    }
  }
  return true;
}
if(chfldl[back] < MAX) {
    if(chfldl[back] == last) buffer[i--] = '1';
} else {
    if(chfldl[back]-MAX == last) buffer[i--] = '1';
}
last = back;
back = parindx[back];

if(chfl0[back] < MAX) {
    if(chfl0[back] == last) buffer[i--] = '0';
} else {
    if(chfl0[back]-MAX == last) buffer[i--] = '0';
}

if(chfldl[back] < MAX) {
    if(chfldl[back] == last) buffer[i--] = '1';
} else {
    if(chfldl[back]-MAX == last) buffer[i--] = '1';
}

// now output the code based on the 0s & 1s in the tmp buffer
// fprintf(test," ");
for(i=0; buffer[i]; i++) {
    // fprintf(test,"%c",buffer[i]);
    if(buffer[i]==1) outbyte |= (1<<outbpos);
    outbpos++;
    if(outbpos==8) {putc(outbyte,fp); outbyte=0; outbpos=0;}
}
// fprintf(test,

Xptr++;
}

// output leftover byte
if(outbpos>0) putc(outbyte,fp);
    return true;
}

/*
 unhuff15.cpp
*/

#include "huff15.h"

// function prototypes
bool buildtree(FILE *fp);
bool decodesymbols(FILE *fp, unsigned int huge *X, long numX);
static void traverse(unsigned int node);

static int inbpos; // global variable for output byte position
static int inbyte; // global variable for output byte

static unsigned int *parent; // global index to parent table
static unsigned int *child0; // global zero child table
static unsigned int *child1; // global one child table

//static FILE *test;

bool unhuffl5(FILE *fp, unsigned int huge *X)
{
    unsigned int i;
    long numX;

    // Testing
    // test = fopen("test.unh","w");

    // read in first 4 bytes to determine the number of X values to be decoded
    numX = 0;
    for(i=0; i<4; i++) {
        if(i>0) numX « =  8;
        inbyte = getc(fp);
        if(inbyte==EOF) return false;
        numX += inbyte;
        fprintf(test,"%d
",inbyte);
    }

    // Testing
    // fprintf(test,"numX=%ld\n",numX);
    // fclose(test);

    // allocate space for Huffman tree information
    parent = (unsigned int *) calloc(MAX,sizeof(unsigned int));
    if(parent==NULL) {
        return false;
    }
    child0 = (unsigned int *) calloc(MAX,sizeof(unsigned int));
    if(child0==NULL) {
        free(parent);
        return false;
    }
child1 = (unsigned int *) calloc(MAX, sizeof(unsigned int));
if(child1 == NULL) {
    free(parent);
    free(child0);
    return false;
}

// initialize tables
for(i=0; i<MAX; i++) {
    parent[i] = 2*MAX+1;
    child0[i] = MAX;
    child1[i] = MAX;
}

// initialize input variables
inbyte = getc(fp);
inbpos = 0;

// testing
// fprintf(test, "Huffman Tree Info:\n");

// build tree from input file
if(!buildtree(fp)) {
    free(parent);
    free(child0);
    free(child1);
    return false;
}

// testing
// fprintf(test, "\nTraversing:\n");
// traverse(0);

// decode symbols from input file into X
if(!decodesymbols(fp, X, numX)) {
    free(parent);
    free(child0);
    free(child1);
    return false;
}

free(parent);
free(child0);
free(child1);
// fclose(test);

    return true;
}

// Build Huffman tree from header information
bool buildtree(FILE *fp)
{
    unsigned int current, next, symbol;
    int i, countzero;

    current = 0;
    next = 1;
    while(true) {

        // count number of zeros on input
        countzero = 0;
        while(!(inbyte & (1<<inbpos))) {
            inbpos++;
            if(inbpos==8) { // read in next input byte if necessary
                inbpos = 0;
                inbyte = getc(fp);
                if(inbyte==EOF) return false;
            }
            countzero++;
            // fprintf(test, "0");
        }
        // skip past 1-bit
        inbpos++;
        if(inbpos==8) { // read in next input byte if necessary
            inbpos = 0;
            inbyte = getc(fp);
            if(inbyte==EOF) return false;
        }

        // follow the left-most path of zeros (must be more than 1 zero)
        for(i=1; i<countzero; i++){
            child0[current] = MAX + next;
            parent[next] = current;
            current = next;
            next++;
        }

        // read in the symbol

    }

    return true;
}
symbol = 0;
for(i=0; i<NUMBITS; i++) {
    if(inbyte & (1<<inbpos))
        symbol += (1<<i);
inbpos++;
    if(inbpos==8) { // read in next input byte if necessary
        inbpos = 0;
inbyte = getc(fp);
        if(inbyte==EOF) return false;
    }
}
// fprintf(test,"l - %u\n",symbol);

// assign symbol to proper child - 0 or 1
if(countzero > 0) child0[current] = symbol;
else child1[current] = symbol;

// go back up tree until next available right branch can be taken
while(child1[current]!=MAX && current>0)
    current = parent[current];

// check to see if no more right branches to be taken
if(child1[current]!=MAX) return true;

// if next bit is not a 1, then make a right branch and go there
if(!(inbyte & (1<<inbpos))) {
    child1[current] = MAX + next;
    parent[next] = current;
    current = next;
    next++;
}

// Decode symbols into X
bool decodesymbols(FILE *fp, unsigned int huge *X, long numX)
{
    unsigned int huge *Xptr, current, symbol;

    Xptr = X;
    while(numX-- > 0) { // decode symbols 1 at a time
        // start at the root and traverse tree for each 0/1 input
current = 0;
symbol = MAX;
while(symbol>=MAX) {
    if((inbyte & (1<<inbpos++)) // child 1
        symbol = child1[current];
    else // child 0
        symbol = child0[current];

    if(symbol<MAX) {  // if symbol, assign it to X and continue
        *Xptr++ = symbol;
        if(inbpos==8 & & numX>0) { // read in next input byte if necessary
            inbpos = 0;
            inbyte = getc(fp);
            if(inbyte==EOF) return false;
        }
    } else {  // not symbol, keep traversing
        current = symbol - MAX;
        if(inbpos==8) { // read in next input byte if necessary
            inbpos = 0;
            inbyte = getc(fp);
            if(inbyte==EOF) return false;
        }
    }
}
}

return true;

void traverse(unsigned int node)
{
    if(child0[node] > MAX) {
        // fprintf(test,"0");
        traverse(child0[node]-MAX);
    } else if(child0[node] < MAX) {
        // fprintf(test,"0");
        // fprintf(test,"1 - %u\n", child0[node]);
    }
    if(child1[node] > MAX) {
        // fprintf(test,"1 - %u\n", child1[node]);
        traverse(child1[node]-MAX);
    } else if(child1[node] < MAX) {
        // fprintf(test,"1 - %u\n", child1[node]);
    }
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Adaptive Huffman Encoding

/*
ah.h
*/

#ifndef __AH_H
#define __AH_H

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

#define MAXBITS 10
#define MAXSYMBOLS (1<<MAXBITS)+3
#define ESCAPE MAXSYMBOLS-3
#define ENDOFFILE MAXSYMBOLS-2
#define ROOT MAXSYMBOLS-1
#define BUFFERSIZE 1024

/* Huffman tree structure */
typedef struct huffman NODE;
struct huffman {
    NODE *next;
    NODE *prev;
    NODE *zero;
    NODE *one;
    NODE *parent;
    unsigned long freq;
};

bool huffman(FILE *fp, unsigned int huge *data, long datalen);
bool huffman(FILE *fp, unsigned char huge *data, long datalen);

#endif

/*
Filename: ah.cpp - Basic N-bit adaptive huffman coding program
Date: 4/1997
Author: Dan Jenkins
*/
#include "ah.h"

/* prototypes */
static bool initialize(void);
static void writecode(FILE *fp, unsigned int);
static bool addcode(unsigned int);
static void update(unsigned int);

NODE **table, *first;
static int escape, pos, BITS;
static unsigned char current;

/*
   Adaptive Huffman coding function.
   Reads the input data and writes the huffman codes to the file fp.
*/
bool huffman(FILE *fp, unsigned int huge *data, long datalen)
{
    unsigned int code;
    unsigned int huge *ptr,
    BITS = 10;

    /* Initialize huffman tree to contain ROOT, ESCAPE, and EOF symbols. */
    if(!initialize()) return false;

    ptr = data;
    while(datalen-- > 0) {
        code=(unsigned int)*ptr-;
        if(table[code]!=(NODE *)0) /* Symbol already in tree */
            writecode(fp,code);
        else { /* Symbol not in tree */
            writecode(fp,ESCAPE);
            writecode(fp,code);
            if(!addcode(code)) return false;
        }
        update(code);
    }

    /* Output end of file code */
    writecode(fp,ENDOFFILE);

    return true;
}
bool huffman(FILE *fp, unsigned char huge *data, long datalen)
{
    unsigned int code;
    unsigned char huge *ptr;

    BITS = 10;
    /* Initialize huffman tree to contain ROOT, ESCAPE, and EOF symbols. */
    if(!initializeO) return false;

    ptr = data;
    while(datalen-- > 0) {
        code=(unsigned int)*ptr++;
        if(table[code]!=(NODE *)0) /* Symbol already in tree */
            writecode(fp,code);
        else { /* Symbol not in tree */
            writecode(fp,ESCAPE);
            writecode(fp,code);
            if(!addcode(code)) return false;
        }
        update(code);
    }
    /* Output end of file code */
    writecode(fp,ENDOFFILE);
    return true;
}

/*
   This function will initialize the huffman tree to contain the EOF code
   and the ESCAPE code only. The decoding routine starts out the same way.
   */
bool initialize(void)
{
    if((table=(NODE **) faralloc(MAXSYMBOLS, sizeof(NODE *)))==NULL)
        return false;
    if((table[ESCAPE]=(NODE *) malloc(sizeof(NODE))) == NULL)
        return false;
}
if((table[ROOT] = (NODE *) malloc(sizeof(NODE))) == NULL)
    return false;

if((table[ENDOFFILE] = (NODE *) malloc(sizeof(NODE))) == NULL)
    return false;

/* Initialize the ESCAPE code in the tree. */
table[ESCAPE]->next = table[ENDOFFILE];
table[ESCAPE]->prev = (NODE *)0;
table[ESCAPE]->zero = (NODE *)0;
table[ESCAPE]->one = (NODE *)0;
table[ESCAPE]->parent = table[ROOT];
table[ESCAPE]->freq = (unsigned long)1;

/* Initialize the ENDOFFILE code in the tree. */
table[ENDOFFILE]->next = table[ROOT];
table[ENDOFFILE]->prev = table[ESCAPE];
table[ENDOFFILE]->zero = (NODE *)0;
table[ENDOFFILE]->one = (NODE *)0;
table[ENDOFFILE]->parent = table[ROOT];
table[ENDOFFILE]->freq = (unsigned long)1;

/* Initialize the ROOT for the tree. */
table[ROOT]->next = (NODE *)0;
table[ROOT]->prev = table[ENDOFFILE];
table[ROOT]->zero = table[ESCAPE];
table[ROOT]->one = table[ENDOFFILE];
table[ROOT]->parent = (NODE *)0;
table[ROOT]->freq = (unsigned long)2;

first = table[ESCAPE];

// initialize output variables
escape = pos = 0;
current = 0;

return true;
}

bool addcode(unsigned int code)
{
    NODE *newparent;

    /* make space for new parent node */
if((newparent=(NODE *) malloc(sizeof(NODE))) == NULL)
    return false;

/* make space for new code */
if((table[code]=(NODE *) malloc(sizeof(NODE))) == NULL)
    return false;

/* Initialize the new parent node of this code in the tree. */
first->parent->zero = newparent;
newparent->next = first->next;
newparent->prev = first;
newparent->zero  = table[code];
newparent->one = first;
newparent->parent = first->parent;
newparent->freq = first->freq;

/* Initialize the new code in the tree. */
table[code]->next = first;
table[code]->prev = (NODE *)0;
table[code]->zero = (NODE *)0;
table[code]->one = (NODE *)0;
table[code]->parent = newparent;
table[code]->freq = (unsigned long)0;

/* update the previous first code info */
first->next = newparent;
first->prev = table[code];
first->parent = newparent;

/* now set the global pointer first equal to the new code */
first = table[code];

return true;
}

void writecode(FILE *fp, unsigned int code)
{
    unsigned int stacksize=0, i, j;
    NODE *ptr;
    char codestack[BUFFERSIZE];
    int numbytes;

    /* If last code written was the escape code then write this code just like
     it is, with the appropriate number of bits, so the decoding program can
add it to its huffman tree. */
if(escape) {
  for(i=0; i<BITS; i++) {
    if((unsigned int)(1<<i) & code) codestack[i]='1';
    else codestack[i]='0';
  }
  codestack[i]='0';
  escape=0; /* reset the escape flag */
} else {
  /* find out how big to make the code stack and allocate the space */
  for(ptr=table[code]; ptr->parent!=(NODE *)0; ptr=ptr->parent) stacksize++;

  /* put the chars '0' or '1' into the codestack in reverse order. */
  codestack[stacksize--] = '0'; /* mark the stopping point */
  for(ptr=table[code]; ptr->parent!=(NODE *)0; ptr=ptr->parent) {
    if(ptr->parent->zero == ptr)
      codestack[stacksize--] = '0';
    else
      codestack[stacksize--] = '1';
  }
}

/* now set the proper bits in the output buffer to be written to the file */
for(i=0; codestack[i]; i++) {
  if(codestack[i]=='1') current |= (1<<pos);
  pos--;
  if(pos==8){
    putc(current,fp);
    pos=0;
    current=0;
  }
}

/* Testing... print out the code
  printf("%s\n",codestack);
  */

/* Handle the special cases ESCAPE and ENDOFFILE */

/* set escape to 1 so next time we will output the code directly */
if(code==ESCAPE) escape=1;

/* Flush the output buffer, cuz we are done! */
if(code==ENDOFFILE) {
if(pos>0) putc(current,fp);
}
}

/*
 * Increments the frequency of the code and updates the huffman tree accordingly
 */
void update(unsigned int code)
{
    NODE *ptr1, *ptr2, *ptr1prev, *ptr2prev, *tmp;

    for(ptr1=table[code]; ptr1!=(NODE *)0; ptr1=ptr1->parent) {
        ptr1->freq = ptr1->freq+1; /* update frequency of this code */
    }

    /* check if need to swap nodes */
    if(ptr1->parent!=(NODE *)0 && ptr1->freq > ptr1->next->freq) {
        /* find the last node->next->freq < ptr1->freq to swap with */
        for(ptr2=ptr1->next; ptr1->freq > ptr2->next->freq; ptr2=ptr2->next);

        /* swap ptr1 with ptr2 */
        /* swap the prev/next links */

        /* if ptr1 is the first element in the list */
        if(ptr1==first) {
            first=ptr2;
        } else {
            ptr1->prev->next = ptr2;
            ptr2->prev->next = ptr1;
            tmp = ptr1->prev;
            ptr1->prev = ptr2->prev;
            ptr2->prev = tmp;
            ptr1->next = ptr2->next;
            ptr2->next = tmp;
        }
    } else {

    }

    if(ptr1->next != ptr2) { /* ptr1 not next to ptr2 */
        ptr1->next->prev = ptr2;
        ptr2->prev->next = ptr1;
        tmp = ptr1->prev;
        ptr1->prev = ptr2->prev;
        ptr2->prev = tmp;
        ptr1->next = ptr2->next;
        ptr2->next = tmp;
    } else {

}
ptr2->next->prev = ptr1;
ptr2->prev = ptr1->prev;
ptr1->prev = ptr2;
ptr1->next = ptr2->next;
ptr2->next = ptr1;
}

/* perform swap of parent links */
if(ptr1->parent != ptr2->parent) {
  if(ptr1->parent->zero == ptr1)
    ptr1->parent->zero = ptr2;
  else ptr1->parent->one = ptr2;
  if(ptr2->parent->zero == ptr2)
    ptr2->parent->zero = ptr1;
  else ptr2->parent->one = ptr1;
  tmp = ptr1->parent;
  ptr1->parent = ptr2->parent;
  ptr2->parent = tmp;
} else {
  tmp = ptr1->parent->zero;
  ptr1->parent->zero = ptr1->parent->one;
  ptr1->parent->one = tmp;
}
}

/*
 unah.h
*/

#ifndef __UNAH_H
#define __UNAH_H

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

#define MAXBITS 10
#define MAXSYMBOLS (1<<MAXBITS)+3
#define ESCAPE MAXSYMBOLS-3
#define ENDOFFILE MAXSYMBOLS-2
#define ROOT MAXSYMBOLS-1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
#define BUFFERSIZE 1024

/* Huffman tree structure */
typedef struct unhuffman NODE2;
struct unhuffman {
   NODE2 *next;
   NODE2 *prev;
   NODE2 *zero;
   NODE2 *one;
   NODE2 *parent;
   unsigned long freq;
   unsigned long symbol;
};

bool unhuffman(FILE *fp, unsigned int huge *data);
bool unhuffman(FILE *fp, unsigned char huge *data);

#endif

/* prototypes */
static bool initialize(void);
static unsigned int getcode(FILE *fp, int bits);
static bool addcode(unsigned int);
static void update(unsigned int);
static NODE2 **table, *first;
static unsigned long numsymbols;
static int pos, BITS;
static unsigned char current;

/*
Adaptive Huffman de-coding function.
Reads the input file *fp and writes the huffman codes to the file databuffer.
*/
bool unhuffman(FILE *fp, unsigned int huge *data)
unsigned int code;
unsigned int huge *ptr;

BITS = 10;
numsymbols = MAXSYMBOLS;

/* Initialize huffman tree to contain ROOT, ESCAPE, and EOF symbols. */
if(!initialize()) return false;

ptr = data;
while((code=getcode(fp, 0)) != ENDOFFILE) {
  if(code!=ESCAPE) { /* Symbol already in tree */
    *ptr++ = code;
  } else { /* Symbol not in tree */
    code = getcode(fp, BITS);
    *ptr++ = code;
    if(!addcode(code)) return false;
  }
  update(code);
}

return true;

bool unhuffinan(FILE *fp, unsigned char huge *data)
{
  unsigned int code;
  unsigned char huge *ptr;

  BITS = 10;
  numsymbols = MAXSYMBOLS;

  /* Initialize huffman tree to contain ROOT, ESCAPE, and EOF symbols. */
  if(!initialize()) return false;

  ptr = data;
  while((code=getcode(fp, 0)) != ENDOFFILE) {
    if(code!=ESCAPE) { /* Symbol already in tree */
      *ptr++ = (unsigned char)code;
    } else { /* Symbol not in tree */
      code = getcode(fp, BITS);
      *ptr++ = (unsigned char)code;
      if(!addcode(code)) return false;
    }
  }

  return true;
}
This function will initialize the huffman tree to contain the EOF code and the ESCAPE code only. The decoding routine starts out the same way.

bool initialize(void)
{
    if((table=(NODE2 **) malloc(MAXSYMBOLS, sizeof(NODE2 *)))==NULL)
        return false;
    if((table[ESCAPE]=(NODE2 *) malloc(sizeof(NODE2))) == NULL)
        return false;
    if((table[ROOT]=(NODE2 *) malloc(sizeof(NODE2))) == NULL)
        return false;
    if((table[ENDOFFILE]=(NODE2 *) malloc(sizeof(NODE2))) == NULL)
        return false;

    /* Initialize the ESCAPE code in the tree. */
    table[ESCAPE]->next = table[ENDOFFILE];
    table[ESCAPE]->prev = (NODE2 *)0;
    table[ESCAPE]->zero = (NODE2 *)0;
    table[ESCAPE]->one = (NODE2 *)0;
    table[ESCAPE]->parent = table[ROOT];
    table[ESCAPE]->freq = (unsigned long)1;
    table[ESCAPE]->symbol = (unsigned long)ESCAPE;

    /* Initialize the ENDOFFILE code in the tree. */
    table[ENDOFFILE]->next = table[ROOT];
    table[ENDOFFILE]->prev = table[ESCAPE];
    table[ENDOFFILE]->zero = (NODE2 *)0;
    table[ENDOFFILE]->one = (NODE2 *)0;
    table[ENDOFFILE]->parent = table[ROOT];
    table[ENDOFFILE]->freq = (unsigned long)1;
    table[ENDOFFILE]->symbol = (unsigned long)ENDOFFILE;

    /* Update the tree. */
    update(code);
}
return true;
}
/* Initialize the ROOT for the tree. */

    table[ROOT]->next = (NODE2 *)0;
    table[ROOT]->prev = table[ENDOFFILE];
    table[ROOT]->zero = table[ESCAPE];
    table[ROOT]->one = table[ENDOFFILE];
    table[ROOT]->parent = (NODE2 *)0;
    table[ROOT]->freq = (unsigned long)2;
    table[ROOT]->symbol = (unsigned long)ROOT;

first = table[ESCAPE];

    pos=8;
    current=0;

    return true;
}

/*
This is the input routine. It's written to input hufnan codes from the
adaptive huffman tree, or if bits>0 then the next code read is built from
the next bits in the input stream. It could be up to 16-bits.
*/

unsigned int getcode(FILE *fp, int bits)
{
    int i, j;
    unsigned int code=0;
    NODE2 *ptr;
    int tmpch;

    if(pos==8) {  // initial case
        tmpch = getc(fp);
        if(tmpch=EOF) return code;
        else current = tmpch;
        pos=0;
    }

    if(bits>0) { /* get a fixed number of bits */
        i=0;
        while(i<bits) {
            if(current & (1«pos++)) { code |= ((unsigned int)(1«i)); }
            i++;
            if(pos==8) {
                tmpch = getc(fp);
                if(tmpch==EOF) return code;
            }
        }
    }
}
else current = tmpch;
pos=0;
}
}
}

}  /* traverse the AH tree and find the code */
ptr = table[ROOT];
while(ptr->zero!=(NODE2 *)0 && ptr->one!=(NODE2 *)0) {
  if(current & (1<<pos++)) { ptr=ptr->one; }
  else {  ptr=ptr->zero; }
  if(pos==8) {
    tmpch = getc(fp);
    if(tmpch==EOF) return ENDOFFILE;
    else current = tmpch;
pos=0;
  }
}
}

code = ptr->symbol;
return(code);

bool addcode(unsigned int code)
{
  NODE2 *newparent;

  /* make space for new parent node */
  if((newparent=(NODE2 *) malloc(sizeof(NODE2))) == NULL)
    return false;

  /* make space for new code */
  if((table[code]=(NODE2 *) malloc(sizeof(NODE2))) == NULL)
    return false;

  /* Initialize the new parent node of this code the tree. */
  first->parent->zero = newparent;
  newparent->next = first->next;
  newparent->prev = first;
  newparent->zero = table[code];
  newparent->one = first;
  newparent->parent = first->parent;
  newparent->freq = first->freq;
  newparent->symbol = numsymbols++;

  /* Initialize the new code in the tree. */
table[code]->next = first;
table[code]->prev = (NODE2 *)0;
table[code]->zero = (NODE2 *)0;
table[code]->one = (NODE2 *)0;
table[code]->parent = newparent;
table[code]->freq = (unsigned long)0;
table[code]->symbol = (unsigned long)code;

/* update the previous first code info */
first->next = newparent;
first->prev = table[code];
first->parent = newparent;

/* now set the global pointer first equal to the new code */
first = table[code];
return true;
}

/*
    Increments the frequency of the code and updates the huffman tree
    accordingly
*/
void update(unsigned int code)
{
    NODE2 *ptr1, *ptr2, *ptr1prev, *ptr2prev, *tmp;

    for(ptr1=table[code]; ptr1!=(NODE2 *)0; ptr1=ptr1->parent) {
        ptr1->freq = ptr1->freq+1; /* update frequency of this code */
    }

    /* check if need to swap nodes */
    if(ptr1->parent!=(NODE2 *)0 && ptr1->freq > ptr1->next->freq) {
        /* find the last node->next->freq < ptr1->freq to swap with */
        for(ptr2=ptr1->next; ptr1->freq > ptr2->next->freq; ptr2=ptr2->next);

        /* swap ptr1 with ptr2 */

        /* swap the prev/next links */

        /* if ptr1 is the first element in the list */
        if(ptr1==first) {
            first=ptr2;
        } else {
            ptr1->prev->next = ptr2;
        }
    }
}
if(ptr1->next != ptr2) { /* ptr1 not next to ptr2 */
    ptr1->next->prev = ptr2;
    ptr2->prev->next = ptr1;
    ptr2->next->prev = ptr1;
    tmp = ptr1->prev;
    ptr1->prev = ptr2->prev;
    ptr2->prev = tmp;
    tmp = ptr1->next;
    ptr1->next = ptr2->next;
    ptr2->next = tmp;
} else {
    ptr2->next->prev = ptr1;
    ptr2->prev = ptr1->prev;
    ptr1->prev = ptr2;
    ptr1->next = ptr2->next;
    ptr2->next = ptr1;
}

/* perform swap of parent links */
if(ptr1->parent != ptr2->parent) {
    if(ptr1->parent->zero == ptr1)
        ptr1->parent->zero = ptr2;
    else ptr1->parent->one = ptr2;
    if(ptr2->parent->zero == ptr2)
        ptr2->parent->zero = ptr1;
    else ptr2->parent->one = ptr1;
    tmp = ptr1->parent;
    ptr1->parent = ptr2->parent;
    ptr2->parent = tmp;
} else {
    tmp = ptr1->parent->zero;
    ptr1->parent->zero = ptr1->parent->one;
    ptr1->parent->one = tmp;
}

LZ77 Encoding

/
L77.H

*/

#ifndef __L77_H
#define __L77_H

#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<string.h>

bool L77(FILE *fpw, long numpixels, unsigned char huge *pixels);
#endif

/*
  Filename: L77.cpp - LZ77 sliding window dictionary compression program
  (window size=12bits/4096bytes, look ahead=4bits/17bytes)
  Date: 3/1997
  Author: Dan Jenkins
*/

#include "l77.h"

/* constants */
#define WINBITS 12        /* bits to code window positions */
#define WINBYTES (1<<WINBITS) /* number of bytes in window buffer */
#define UNUSED (WINBYTES+511) /* denoting an unused pointer in btree */
#define WINMASK (WINBYTES-1) /* window buffer index mask */
#define LABITS 4           /* bits to code length of match
                        NOTE: this is large because huffman coding will be applied afterwards to shrink it down */
#define LBYTES ((1<<LABITS)+1) /* number of bytes in lookahead buffer */
#define MINMATCH 2         /* minimum number of bytes to make a match
                        NOTE: this may look small compared to the size of LABITS, but its related to later huffman coding as well. (at least I hope) */
#define LABUFFSIZE 4096     /* Lookahead buffer size */

/* function prototypes */
static bool initl77(void);
static unsigned int longest(unsigned char [], int, int &);
static void addstring(unsigned char [], int);
static void delstring(unsigned int);
static int compare(unsigned char [], int, int);
static void writecode77(int, int, int);

/* Binary tree structure, tree[WINBYTES]=root */
struct node {
   unsigned int parent;
   unsigned int lt;
   unsigned int gt;
};

/* Global variables */
static FILE *fpw; /* output file ID */
static struct node *btree; /* binary tree */
static unsigned char *winbuffer; /* window buffer */
static unsigned int next; /* next available index in the window buffer */
static unsigned int ROOT;

LZ77 coding function. Follows the general algorithm pretty close except
that it uses one bit to tell the difference between a "non-match" (0),
output of just one character and a "match" (1), output of a window index
and length of match pair.

Reads the input file "fin" and writes the huffman codes to the file "finw".
*
bool L77(FILE *outfile, long numpixels, unsigned char huge *pixels)
{
   int i, j, numbytes, length, offset=0, tmpx, tmpc;
   unsigned int matchndx, tmp;
   unsigned char labuffer[LABUFFSIZE]; /* lookahead buffer */
   unsigned char huge *pix;
   unsigned char *head, ch;
   long numcodes;

   next=0; /* next available index in the window buffer */

   if(!initl77()) return false;

   fpw = outfile;

   // Add the compression method after the file header
   putc('L',fpw);
   putc('7',fpw);
   putc('7',fpw);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
numcodes = numpixels;
pix = pixels;

/* first fill the LA buffer and initialize window buffer */
for(i=0; i<LAYBYTES && numcodes>0; i++) {
    ch = *pix++;
    numcodes--;
    winbuffer[i] = labuffer[i] = ch;
}

numbytes = i;

while(numbytes>0) {
    /* if match found output index and length */
    if(numbytes >= MINMATCH &&
        (matchndx=longest(labuffer+offset,numbytes,length))!=UNUSED){
        tmpc = length-MINMATCH;
        tmpx = next-matchndx;
        if(tmpx < 0) tmpx = next+(WINBYTES-matchndx);
        writecode77(l, tmpc, tmpx);
        if((offset+numbytes+length) >= LABUFFSIZE) {
            /* shift the look ahead string to front of buffer */
            for(i=0; i<numbytes; i++)
                labuffer[i] = labuffer[i+offset];
            offset=0;
        }
        for(j=0; j<length; j++) {
            //printf("%d - %d\n",j+1,winbuffer[(matchndx+j)&WINMASK]);
            addstring(labuffer+offset, numbytes);
            if(numcodes==0) { /* getting towards the end */
                numbytes--;
            } else {
                ch = *pix++;
                numcodes--;
                /* add new char to look ahead buffer */
                labuffer[offset+numbytes]=ch;
                /* delete the old string at this position in winbuffer */
                tmp=(next+numbytes)&WINMASK;
                delstring(tmp);
                /* add the new character to the winbuffer */
                winbuffer[tmp]=ch;
            }
        }
    /* adjust the look ahead buffer offset */
    offset++;
    next = (next+1)&WINMASK;
    }
else { /* not a match, output single character */
    // printf("0 - %d\n",labuffer[offset]);

    /* write code to output file */
    writecode77(0, (int)&buffer[offset], 0);
    addstring(labuffer+offset, numbytes);
    if((offset+numbytes+1) >= LABUFFSIZE) {
        /* shift the look ahead string to front of buffer */
        for(i=0; i<numbytes; i++)
            labuffer[i] = labuffer[i+offset];
        offset=0;
    }
    if(numcodes==0) { /* getting towards the end */
        numbytes--;
    } else {
        ch = *pix++;
        numcodes--;
        /* add new char to look ahead buffer */
        labuffer[offset+numbytes]=ch;
        /* delete the old string at this position in winbuffer */
        tmp=(next+numbytes)&WINMASK;
        delstring(tmp);
        /* add the new character to the winbuffer */
        winbuffer[tmp]=ch;
    }
    /* adjust the look ahead buffer offset */
    offset++;
    next = (next+1)&WINMASK;
}

/* write the end of file signal */
writecode77(1,0,0);

/* Close the files and exit */
return true;
}

/* Initializes binary tree structure */
bool init77(void)
{
    unsigned int i;
btree=(struct node *)malloc(sizeof(struct node)*(WINBYTES+511));
if(btree==NULL) return false;

if((winbuffer=(unsigned char *)malloc(sizeof(unsigned char)*WINBYTES))==NULL)
return false;

for(i=0; i<WINBYTES; i++) {
    btree[i].parent = UNUSED;
    btree[i].lt = UNUSED;
    btree[i].gt = UNUSED;
    winbuffer[i] = 0;
}

/* extra nodes for ROOT */
for(; i<WINBYTES+511; i++){
    btree[i].parent = UNUSED;
    btree[i].lt = UNUSED;
    btree[i].gt = UNUSED;
}

return true;

/*
By using the binary search tree it searches for the longest string match in
the window buffer to match up to num bytes of the string starting at Indx
in the lookahead buffer. It returns the index of the string in the window
buffer only if the longest match is not less than the MINMATCH amount,
otherwise it returns 0.

NOTE: Could be a pain in the ass.
*/
unsigned int longest(unsigned char la[], int num, int &max)
{
    int count=0, maxptr=0, lx, wx;
    unsigned int ptr=0;
    // printf("Begin - longest(%d,%d)\n",num,max);

    max = 0;
    ROOT = WINBYTES + (unsigned int)la[0] + (unsigned int)la[1];
    ptr = btree[ROOT].lt; /* index past the root of the tree to first node */
    while(ptr!=UNUSED && & max<num && count<=max){
        lx = 0;
        if(winbuffer[ptr]==la[lx]) { /* onebyte matches */
/* count how many bytes match */
  count=1;
  wx = ptr;
  while(count<num && winbuffer[++wx&WINMASK]==la[++lx]){  
    count++; 
  }
  if(count>max) { max=count; maxptr=ptr; }
  if(winbuffer[wx&WINMASK] > la[lx])
    ptr = btree[ptr].lt;
  else ptr = btree[ptr].gt;
  } else if(winbuffer[ptr] > la[lx]) {
    ptr = btree[ptr].lt;
  } else {
    ptr = btree[ptr].gt;
  }
  // printf("End - longest(%d,%d)\n",num,max);

  if(max<MINMATCH) return(UNUSED);
  else return(maxptr);

  /*
  Adds the string (up to num bytes) at Indx in the lookahead buffer to the
  window buffer at the next available window position, updating the binary
  search tree as well.
  
  NOTE: This is going to be a BITCH!
  */
  void addstring(unsigned char la[], int num)
  {
    int cmp=0, found=0;
    unsigned int ptr;
    /* first delete the old string at the next available position */
    // delstring(next);
    // printf(" Begin - addstring(%d)\n",num);
    /* search for the position in the tree to add the new string */
    ROOT = WINBYTES + (unsigned int)la[0] + (unsigned int)la[1];
    ptr = btree[ROOT].lt;
    while(ptr!=UNUSED && !found) {
      cmp = compare(la, ptr, num);
if(cmp==0) {
    found=1;
} else if(cmp<0) {
    if(btree[ptr].lt == UNUSED) found=1;
    else ptr = btree[ptr].lt;
} else if(cmp>0) {
    if(btree[ptr].gt == UNUSED) found=1;
    else ptr = btree[ptr].gt;
}

if(cmp==0) { /* replace current node with new one */
    if(ptr==UNUSED) { /* initial case */
        btree[ROOT].lt = next;
        btree[next].parent = ROOT;
        btree[next].lt = UNUSED;
        btree[next].gt = UNUSED;
    } else {
        if(ptr == btree[btree[ptr].parent].lt)
            btree[btree[ptr].parent].lt = next;
        else
            btree[btree[ptr].parent].gt = next;
        btree[next].lt = btree[ptr].lt;
        btree[next].gt = btree[ptr].gt;
        btree[next].parent = btree[ptr].parent;
        if(btree[ptr].lt != UNUSED) btree[btree[ptr].lt].parent = next;
        if(btree[ptr].gt != UNUSED) btree[btree[ptr].gt].parent = next;
        /* delete old node */
        btree[ptr].parent = UNUSED;
        btree[ptr].lt = UNUSED;
        btree[ptr].gt = UNUSED;
    }
} else {
    btree[next].lt = UNUSED;
    btree[next].gt = UNUSED;
    btree[next].parent = ptr;
    if(cmp<0) { /* replace the LT node */
        btree[ptr].lt = next;
    } else if(cmp>0) { /* replace the GT node */
        btree[ptr].gt = next;
    }
}

// printf(" End - addstring(%d)\n",num);
}
/* Delete string at this node in btree */
void delstring(unsigned int node)
{
    unsigned int ptr;

    // printf(" Begin - delstring(%d)\n",node);

    /* nothing to do if not attached to start with */
    if(btree[node].parent==UNUSED){
        // printf(" End - delstring(%d)\n",node);
        return;
    }

    if(btree[node].lt==UNUSED && btree[node].gt==UNUSED) { /*both children null*/
        if(node == btree[btree[node].parent].lt)
            btree[btree[node].parent].lt = UNUSED;
        else
            btree[btree[node].parent].gt = UNUSED;
    } else if(btree[node].lt==UNUSED || btree[node].gt==UNUSED) { /* one child null */
        if(node == btree[btree[node].parent].lt) {
            if(btree[node].lt!=UNUSED) {
                btree[btree[node].parent].lt = btree[node].lt;
                btree[btree[node].lt].parent = btree[node].parent;
            } else {
                btree[btree[node].parent].lt = btree[node].gt;
                btree[btree[node].gt].parent = btree[node].parent;
            }
        } else {
            if(btree[node].lt!=UNUSED) {
                btree[btree[node].parent].gt = btree[node].lt;
                btree[btree[node].lt].parent = btree[node].parent;
            } else {
                btree[btree[node].parent].gt = btree[node].gt;
                btree[btree[node].gt].parent = btree[node].parent;
            }
        }
    } else { /* both children non-null */
        ptr = btree[node].lt;
        while(btree[ptr].gt!=UNUSED) ptr = btree[ptr].gt; /* find replacement node */
        delstring(ptr); /* ptr and delete it */

        /* now replace node with ptr */
        if(node == btree[btree[node].parent].lt)
btree[btree[node].parent].lt = ptr;
else
  btree[btree[node].parent].gt = ptr;
btree[ptr] = btree[node];
btree[btree[ptr].lt].parent = ptr;
btree[btree[ptr].gt].parent = ptr;
}

btree[node].parent = UNUSED;
// printf(" End - delstring(%d)\n",node);
}

int compare(unsigned char la[], int wx, int num)
{
  int i, neq=0, tmp;

  for(i=0; i<num && !neq; i++, wx++)
  {
    tmp = wx&WINMASK;
    if(la[i]<winbuffer[tmp]) { /* less than */
      neq = -1;
    } else if(la[i]>winbuffer[tmp]) { /* greater than */
      neq = 1;
    }
  }

  return(neq);
}

/* Writes the code, ch or len+offset to the output buffer */
void writecodel77(int codetype, int chorlen, int offset)
{
  static unsigned char outtype=0, outdata[24];
  static int otndx=0, odndx=0;
  int i;

  outtype |= (codetype<<otndx++);
  outdata[odndx] = chorlen;

  if(codetype) {
    outdata[odndx++] |= (unsigned char)((offset & 15)<<4);
    outdata[odndx++] = (unsigned char)((offset>>4) & 255);
  } else odndx++;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
/* flush output buffer if full, or at the end of signal*/
if(otndx==8 || (codetype==l && offset==0)) {
    putc(outtype, fpw);
    for(i=0; i<odndx; i++) putc(outdata[i], fpw);
    odndx = odndx = 0;
    outtype = 0;
}

/*
UNLZ77.H
*/

#ifndef __UNL77_H
#define __UNL77_H

#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<string.h>

bool UNL77(FILE *lzwfile, unsigned char huge *pixels);

#endif

/*
Filename: UNL77.cpp - LZ77 sliding window dictionary DEcompression program
(window size=4096 bytes, look ahead=17 bytes)
Date: 3/1997
Author: Dan Jenkins
*/

#include "unl77.h"

/* constants */
#define WINBITS 12 /* bits to code window positions */
#define WINBYTES (1<<WINBITS) /* number of bytes in window buffer */
#define WINMASK (WINBYTES-1) /* Window buffer mask */
#define LABITS 4 /* bits to code length of match
    NOTE: this is large because huffman coding
    will be applied afterwards to shrink it down */
#define LABYTES ((1<<LABITS)+1) /* number of bytes in lookahead buffer */
#define MINMATCH 2 /* minimum number of bytes to make a match
    NOTE: this may look small compared to the
size of LABITS, but its related to later
buhman coding as well. (at least I hope)
*/

/*
   LZ77 DEcoding function. Follows the general algorithm pretty close except
that it uses one bit to tell the difference between a "non-match" (0),
output of just one character and a "match" (1), output of a window index
and length of match pair.
*/

bool UNL77(FILE *input, unsigned char huge *pixels)
{
    int i, j, next=0, matchndx, length, distance, codetype, done=0;
    unsigned char winbuffer[WINBYTES], code;
    unsigned char huge *pix;
    char tmp[10];

    pix = pixels;

    /* Read the first 3 chars of data to be sure it is in L77 compress format */
    for(i=0; i<3; i++) tmp[i]=getc(input);
    tmp[1]=^0;
    if(strcmp(tmp,"L77")!=0) return false;

    while((codetype=getc(input))!=EOF && !done) {
        for(j=0; j<8 && !done; j++) {  
            if(codetype & (1<<j)) { /* output matched codes */
                code = getc(input);
                length = (code & 15) + MINMATCH;
                distance = (code>>,4);
                code = getc(input);
                distance |= (code<<4);
                if((next-distance)>=0) matchndx = next-distance;
                else matchndx = WINBYTES-(distance-next);
                if(distance==0) { /* this is the end, exit gracefully */
                    done=1;
                } else {
                    // printf("Match = %d\n",length);
                    for(i=0; i<length; i++){
                        code = winbuffer[matchndx];
                        winbuffer[next] = code;
                        /*pix++ = code;
                        // printf(" [%d]=%d - [%d]=%d\n",matchndx, winbuffer[matchndx],

```
LZW Encoding

/*
   * LZW.H
   */

#ifndef __LZW_H
#define __LZW_H

#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>

bool LZW(FILE *lzwfile, long numpixels, int newcolors,
        unsigned char huge *pixels);

#endif

/*
   LZW.CPP - Lempel-Ziv-Welch compression class methods
   Dan Jenkins - February, 1997
*/

#include "lzw.h"

static void outputcode(FILE *lzwfile, unsigned int code);
static unsigned int lookup(unsigned int scode, int ch);
static int output_bit_count;
static unsigned long output_bit_buffer;
static int INCREMENT, FLUSH, ENDOFFILE, FIRSTCODE, bits;
static int MAXBITS, MAXVALUE, TOLERANCE;
static int huge *codetable;
static unsigned int huge *stringtable;
static unsigned char huge *chartable;
static long HASHSIZE;

// Image LZW compression method
bool LZW(FILE *lzwfile, long numpixels, int newcolors, unsigned char huge *pixels)
{
    int character, toler=0, i;
    unsigned int index, nextcode, stringcode;
    unsigned char huge *pix;
    unsigned char *head;
    long numcodes;
    int huge *ptr;
    unsigned long hndx;

    HASHSIZE = 65535L;
    MAXBITS=15;
    TOLERANCE=1000;
    MAXVALUE = 32767;

    output_bit_count=0;
    output_bit_buffer=0L;

    codetable = (int huge *) farmalloc(HASHSIZE*sizeof(int)+1);
    stringtable= (unsigned int huge *) farmalloc(HASHSIZE*sizeof(unsigned int)+1);
    chartable = (unsigned char huge *) farmaUoc(HASHSIZE*sizeof(unsigned char)+1);

    if(codetable==NULL || stringtable==NULL || chartable==NULL)
        return false;

    ptr = codetable;
    for(hndx=0; hndx<=HASHSIZE; hndx++) *ptr++ = -1;

    // use number of colors to setup other important values
    INCREMENT = newcolors;
    FLUSH = INCREMENT+1;
    ENDOFFILE = FLUSH+1;
    FIRSTCODE = ENDOFFILE+1;
    nextcode = FIRSTCODE;

for(bits=0, i=nextcode; i>0; bits++) i >>= 1; // set the initial output bits

// add the compression method to the file after the header
putc('L',lzwfile);
putc('Z',lzwfile);
putc('W',lzwfile);

numcodes = numpixels;
pix = pixels;
stringcode = (unsigned int)*pix++;
numcodes--;
while(numcodes > 0) {

    character = (int)*pix++; 
    numcodes--;

    index = lookup(stringcode,character);
    if (*(codetable+index) != -1) {
        stringcode=(unsigned int)*(codetable+index);
        if (nextcode > MAXVALUE && toler < TOLERANCE) toler += 2;
    } else {
        if(nextcode <= MAXVALUE) {
            *(codetable+index)=nextcode++;
            *(stringtable+index)=stringcode;
            *(chartable+index)=character;
        } else {
            if(TOLERANCE < 1000) toler--;
            if(toler < -TOLERANCE) { /* flush code tables and start over */
                // printf("\nFlushing");
                outputcode(lzwfile, stringcode);
                outputcode(lzwfile, FLUSH);
                nextcode = FIRSTCODE;
                for(bits=0, i=nextcode; i>0; bits++) i >>= 1;
                stringcode=(unsigned int)character;
                if(numcodes<=0) {
                    /* output last code, ENDOFFILE code and exit */
                    outputcode(lzwfile, stringcode);
                    outputcode(lzwfile, ENDOFFILE);
                    outputcode(lzwfile,(unsigned int)0);
                    free(codetable);
                    free(stringtable);
                    free(chartable);
                    return true;
                }
            character = (int)*pix++; 
            }
        }
    }
}
numcodes--;    ptr = codetable;
for(hndx=0; hndx<=HASHSIZE; hndx++) {ptr++;  index = lookup(stringcode,character);
*(codetable+index)=nextcode++;  *(stringtable+index)=stringcode;
*(chartable+index)=character;
toler=0;
}
outputcode(lzwfile, stringcode);
stringcode=(unsigned int)character;
/* output last code, ENDOFILE code and exit */
outputcode(lzwfile, stringcode);
outputcode(lzwfile, ENDOFILE);
outputcode(lzwfile,(unsigned int)0);

farfree(codetable);
farfree(stringtable);
farfree(chartable);
return true;

unsigned int lookup(unsigned int scode, int ch)
{
    unsigned int index=0;
    srand(scode ^ (unsigned int)(ch<<(bits-8)));  // random rehash function
    while(1) {
        index = (unsigned int)2*rand();
        if(*(codetable+index) == -1) return(index);
        if(*(stringtable+index)=scode && *(chartable+index)=(unsigned char)ch)
            return(index);
        index = (index+1) % HASHSIZE;
        if(*(codetable+index) == -1) return(index);
        if(*(stringtable+index)=scode && *(chartable+index)=(unsigned char)ch)
            return(index);
    }
}

void outputcode(FILE *lzwfile, unsigned int code)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
{
/* if necessary increase the bits per code */
if(code > ((1<<bits)-1)) {
	/* first output marker code to notify decompression program of change */
	outputcode(lzwfile, INCREMENT);
	/* increase bits per code */
	bits++;
	/* output original code, with approx. number of bits */
	outputcode(lzwfile, code);
} else {
	output_bit_buffer |= (unsigned long) code « (32-bits-output_bit_count);
	output_bit_count += bits;
	while (output_bit_count >= 8) {
		putc(output_bit_buffer>>24,lzwfile);
		output_bit_buffer <<= 8;
		output_bit_count -= 8;
	}
}

/*
   UNLZW.H
*/

#ifndef __UNLZW_H
#define __UNLZW_H

#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<string.h>

bool UNLZW(FILE *lzwfile, int numcolors, unsigned char huge *pixels);
#endif

/*

Dan Jenkins - 3/97
*/
```c
#include "unlzw.h"
#include <limits.h>

#define MAXVALUE 32767L
#define MAXBITS 15

static unsigned char *decodestring(unsigned char *buffer, unsigned int code);
static unsigned int inputcode(FILE *input);

static int inputbitcount;
static unsigned long inputbitbuffer;
static unsigned int *stringtable;
static unsigned char *chartable;
static int INCREMENT, FLUSH, ENDOFFILE, FIRSTCODE, bits;

bool UNLZW(FILE *lzwfile, int numcolors, unsigned char huge *pixels)
{
    char tmp[10];
    unsigned int nextcode;
    unsigned int newcode;
    unsigned int oidcode;
    unsigned int character,
    int i;
    unsigned char *string, decode[5000];
    unsigned char huge *pix;

    inputbitcount=0;
    inputbitbuffer=OL;

    stringtable= (unsigned int *) farmalloc(MAXVALUE*sizeof(unsigned int)+1);
    chartable = (unsigned char *) farmalloc(MAXVALUE*sizeof(unsigned char)+1);
    if (stringtable==NULL || chartable==NULL) return false;

    pix = pixels;

    /* Read the first 3 chars of data to be sure it is in LZW compress format */
    for(i=0; i<3; i++) tmp[i]=getc(lzwfile);
    tmp[i]=\0;
    if(strcmp(tmp,"LZW")!=0) return false;

    INCREMENT = (int)numcolors;
    FLUSH = INCREMENT+1;
    ENDOFFILE = FLUSH+1;
    FIRSTCODE = ENDOFFILE+1;
```
nextcode = FIRSTCODE;
for(bits=0, i=nextcode; i>0; bits++) i >>= 1; // set the initial output bits

oldcode = inputcode(lzwfile);
character = oldcode;

*pix++ = (unsigned char) oldcode;

while(1) {
    newcode = inputcode(lzwfile);

    /* Check for special codes */
    if(newcode == INCREMENT) { // increase the input bit size
        bits++;
    } else if(newcode == ENDOFFILE) { // all done, exit
        farfree(stringtable);
        farfree(chartable);
        return true;
    } else if(newcode == FLUSH) { // flush code tables and start over
        nextcode = FIRSTCODE;
        for(bits=0, i=nextcode; i>0; bits++) i >>= 1; // set the initial output bits
        oldcode = inputcode(lzwfile);
        character = oldcode;

        *pix++ = (unsigned char) oldcode;
    }
    else {
        if (newcode >= nextcode) { // decode string
            *decode = (unsigned char) character;
            string = decodestring(decode+1, oldcode);
        } else string = decodestring(decode, newcode);

        character = *string;
        while(string >= decode)
            *pix++ = (unsigned char) *string--;

        if (nextcode <= MAXVALUE) { // add the code to the table
            stringtable[nextcode] = oldcode;
            chartable[nextcode+1] = character;
        }
        oldcode = newcode;
    }
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
unsigned char *decodestring(unsigned char *buffer, unsigned int code)
{
    int i=0;

    while (code >= INCREMENT) {
        *buffer++ = chartable[code];
        code=stringtable[code];
        if(++i>=5000) {
            // printf(" Error decoding string!\n");
            // exit(-1);
            return(buffer);
        }
    }
    *buffer=(unsigned char)code;
    return(buffer);
}

unsigned int inputcode(FILE *input)
{
    unsigned int returnvalue;
    unsigned int tmpch;

    while(inputbitcount <= 24) {
        if((tmpch=getc(input))==(unsigned)EOF) tmpch=0;
        inputbitbuffer |= (unsigned long) tmpch «  (24-inputbitcount);
        inputbitcount += 8;
    }
    returnvalue=inputbitbuffer >> (32-bits);
    inputbitbuffer <<= bits;
    inputbitcount -= bits;

    return(returnvalue);
}

Block Encoding

/*
   blockNxN.h
*/

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
#include <alloc.h>
#include <stdio.h>
#include <stdlib.h>

#if defined_BLOCKNXN_H
#define _BLOCKNXN_H
#endif

bool blockNxN(FILE *fp, int huge *X, int W, int H, int N);
bool unblockNxN(FILE *fp, int huge *X, int W, int H, int N);
#endif

/*
 * blockNxN.cpp
 */

#include "blockNxN.h"
#include "ah.h"
#include "unahh"
#include "huff15.h"

#define HOFFSET 10000

bool writeNxN(FILE *fp, int data[], int W, int H, int status, int N);

bool blockNxN(FILE *fp, int huge *X, int W, int H, int N)
{
    int i, j, k, l, *data, hleft;
    int huge *Xptr;

data = (int *) malloc(sizeof(int)*(N*N+1));

    // initialize output function
    if(!writeNxN(fp, data, W, H, 0, N)) return false;

    Xptr = X;
    for(j=0; j<=H-N; j+=N) {
        // go through the data in NxN blocks
        for(i=0; i<W; i++, Xptr++) {
            k = i%N;
            for(l=0; l<N; l++) data[N*k + l] = *(Xptr + l*W);
            if(k==N-1) if(!writeNxN(fp, data, W, H, 1, N)) return false;
        }
        // left over values on row
if(k!=N-1) {
    k++;  
    for(; k<N; k++) {
        for(l=0; l<N; l++) data[N*k + l] = 0;
    }
    if(!writeNxN(fp, data, W, H, 1, N)) return false;
}

Xptr += (N-1)*W;
}
// left over pixels in column
if(j<H) {
    hleft = H-j;
    for(i=0; i=W; i++, Xptr++) {
        k=i%N;
        for(l=0; l<hleft; l++) data[N*k + l] = *(Xptr + l*W);
        for(; l<N; l++) data[N*k + l] = 0;
        if(k==N-1) if(!writeNxN(fp, data, W, H, 1, N)) return false;
    }
    // left over values on row
    if(k!=N-1) {
        for(i++; k<N; k++) {
            for(l=0; l<N; l++) data[N*k + l] = 0;
        }
        if(!writeNxN(fp, data, W, H, 1, N)) return false;
    }
}
// flush buffers to the output
if(!writeNxN(fp, data, W, H, 2, N)) return false;
return true;

bool writeNxN(FILE *fp, int data[], int W, int H, int status, int N) {
    int i, min, max;
    static unsigned char huge *A, huge *Aptr;
    static unsigned char huge *B, huge *Bptr;
    static unsigned int huge *C, huge *Cptr;
    static int Apos, Bpos;
    static long Alen, Blen, Clen;
    long numblocks, numcodes;
    if(status==0) { // initialize global buffers & their pointers
        //...
numblocks = (long)(W/N + W%N) * (long)(H/N + H%N);
numcodes = (long)W * (long)H;

A = (unsigned char huge *) farmalloc(numblocks + 1);
if(A==NULL) return false;
B = (unsigned char huge *) farmalloc((long)(N*N*numblocks/8)+1);
if(B==NULL) { farfree(A); return false; }
C = (unsigned int huge *) farmalloc((long)(numcodes+1)*sizeof(unsigned int));
if(C==NULL) { farfree(A); farfree(B); return false; }

Apos = Bpos = 0;
Alen = Blen = Clen = 0;
Aptr = A;
Bptr = B;
Cptr = C;
*Aptr = *Bptr = 0;

return true;

} else if(status==2) { // output buffers to file
if(Apos>0) Alen++;
if(Bpos>0) Blen++;

// huffman code the A's
if(!huffman(fp, A, Alen)) return false;

// huffman code the B's
if(!huffman(fp, B, Blen)) return false;

// huffman code the C's
if(!huffl5(fp, C, Clen)) return false;

/*testing!
FILE *test;
Aptr = A;
test = fopen("blockA.dat", "wb");
while(Alen-- > 0) putc(*Aptr++,test);
fclose(test);
Bptr = B;
test = fopen("blockB.dat", "wb");
while(Blen-- > 0) putc(*Bptr++,test);
fclose(test);
Cptr = C;
test = fopen("blockC.dat", "w");

while(Clen— > 0) fprintf(test,"%u\n",*Cptr++);
fclose(test);

farfree(A);
farfree(B);
farfree(C);
/*

return true;
*/

// find min and max for N x N set of data
max = -32767;
min = 32767;
for(i=0; i<N*N; i++){
    if(data[i]>max) max = data[i];
    if(data[i]<min) min = data[i];
}

if(min==0 && max==0) { // whole block is all zeros
    Apos++;
    // 0 - in Zero/Non-zero BM
    if(Apos==8) { Aptr++; *Aptr=0; Alen++; Apos=0; } 
}
else { // block is not all zeros
    *Aptr |= (1<<Apos); // 1 - in Zero/Non-zero BM
    Apos++;
    if(Apos==8) { Aptr++; *Aptr=0; Alen++; Apos=0; }
}

// store bitmap representing 0=Zero & 1=Non-zero
for(i=0; i<N*N; i++) {
    if(data[i]) {
        *Bptr |= (1<<Bpos);
        *Cptr++ = HOFFSET + data[i];
        Clen++;
    }
    Bpos++;
    if(Bpos==8) { Bptr++; *Bptr=0; Blen++; Bpos=0; }
}

return true;
}
bool unblockNxF(FILE *fp, int huge *X, int W, int H, int N)
{
    int i, j, k, l;
    int huge *Xptr;
    unsigned char huge *A, huge *Aptr;
    unsigned char huge *B, huge *Bptr;
    unsigned int huge *C, huge *Cptr;
    int Apos, Bpos;
    long numbblocks, numcodes;

    // initialize buffers & their pointers
    numbblocks = (long)(W/N + W%N) * (long)(H/N + H%N);
    numcodes = (long)W * (long)H;

    A = (unsigned char huge *) fannalloc(numblocks + 1);
    if(A==NULL) return false;
    B = (unsigned char huge *) fannalloc((long)(N*N*numblocks/8)+1);
    if(B==NULL) { farfree(A); return false; }
    C = (unsigned int huge *) farmalloc((long)(numcodes+1)*sizeof(unsigned int));
    if(C==NULL) { farfree(A); farfree(B); return false; }

    // unhuff1nan code the A's
    if(!unhuffian(fp, A)) return false;

    // unhuffiman code the B's
    if(!unhuffiman(fp, B)) return false;

    // unhuffiman code the C's
    if(!unhuffiman(fp, C)) return false;

    Xptr = X;
    Apos = Bpos = 0;
    Aptr = A;
    Bptr = B;
    Cptr = C;
    i=j=0;
    while(j<H) {
        while(i<W) {
            if(*Aptr & (1<<Apos)) { // Block NON-zero
                for(k=0; k<N & i<W; k++, i++, Xptr++) {
                    for(l=0; l<N & j+l<H; l++) {
                        ...
                    }
                }
            }
            Aptr++;
            i++;
        }
        Bptr++;
        j++;
    }
}
if(*Bptr \& (1<<Bpos)) { // Element NON-zero
    *(Xptr+l*W) = (int)*CptrH- - HOFFSET;
} else *(Xptr+l*W) = 0; // Element Zero
Bpos++;
    if(Bpos==8) { Bptr++; Bpos=0; }
}

if(l<N) { // Odd pixels in column
    for(l=0; l<N; l++) {
        if(*Bptr & (1<<Bpos)) { // Element NON-zero
            // ERROR - there shouldn't be anything but 0's here
            farfree(A);
            farfree(B);
            farfree(C);
            return false;
        }
    }
    Bpos++;
        if(Bpos==8) { Bptr++; Bpos=0; }
    }
}

if(l<k) && k<N) { // Odd pixels in row
    for(l=0; k<N; k++) {
        for(l=0; l<N; l++) {
            if(*Bptr & (1<<Bpos)) { // Element NON-zero
                // ERROR - there shouldn't be anything but 0's here
                farfree(A);
                farfree(B);
                farfree(C);
                return false;
            }
        }
    Bpos++;
        if(Bpos==8) { Bptr++; Bpos=0; }
    }
}

else { // Block all zeros
    for(k=0; k<N && i<W; k++, i++, Xptr++) {
        for(l=0; l<N && j<l<H; l++)
            *(Xptr+l*W) = 0;
    }
    }

    Apos++;
    if(Apos==8) {
        Aptr++; 
        Apos=0;
    }
Sub Sampling Transform

/*
xsub.h

Subsample transform & inverse transform header file
*/

#ifndef __XSUB_H
#define __XSUB_H

#include<alloc.h>

/*
subsample transform
X = original unchanged image (given)
width = width of original image (given)
height = height of original image (given)
Y = transformed image, (subsample included)
level = number of levels to apply the transform, 0=none, 1=1, ...
*/
void xsub(unsigned char huge *X, int W, int H, int huge *Y, int level);

/*
subsample inverse transform
X = original unchanged image (to be calculated)
void invsub(unsigned char huge *X, int W, int H, int huge *Y, int level);

#include "xsub.h"

void xsub(unsigned char huge *X, int W, int H, int huge *Y, int level)
{
    int i, j, Weven, Heven, width, height, Icount;
    unsigned char huge *Xptr, huge *firstX;
    int huge *Yptr, huge *firstY;
    
    width = W;
    height = H;
    Icount = 1;
    while(level-- > 0) {
        // store values for even width & height
        if(width%2) Weven = 0;
        else Weven = 1;
        if(height%2) Heven = 0;
        else Heven = 1;
    }
Xptr = X;
Yptr = Y;

// go through every pixel of original image
for(j=0; j<height-Heven; j++) {
  firstX = Xptr;
  firstY = Yptr;
  for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
    if(i%2) {  // odd column
      if(j%2) { // odd row
        // Original - 4 pixel "cross" estimate
        *Yptr = *Xptr -
        (*Xptr-lcount) + *(Xptr+lcount) +
        *(Xptr-W*lcount) + *(Xptr+W*lcount))/4;
      } else { // even row
        // Original - 2 pixel "horizontal" estimate
        *Yptr = *Xptr - *(Xptr-lcount) + *(Xptr+lcount))/2;
      }
    } else { // even column
      if(j%2) { // odd row
        // Original - 2 pixel "vertical" estimate
        *Yptr = *Xptr - *(Xptr-W*lcount) +
        *(Xptr+W*lcount))/2;
      } else { // even row (sub-sample of original)
        *Yptr = *Xptr;
      }
    }
  }

  if(Weven) {
    if(j%2) { // odd row
      // Original - 3 pixels from the "cross" estimate
      *Yptr = *Xptr -
      *(Xptr-lcount) + *(Xptr-W*lcount) +
      *(Xptr+W*lcount))/3;
    } else { // even row
      // Original - 1 pixel from the "horizontal" estimate
      *Yptr = *Xptr - *(Xptr-lcount);
    }
  }
}

Xptr = firstX + W*lcount;
Yptr = firstY + W*lcount;

if(Heven) {
  for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
    if(i%2) { // odd column
      // Original - 2 pixel "vertical" estimate
      *Yptr = *Xptr - *(Xptr-W*lcount) +
      *(Xptr+W*lcount))/2;
    } else { // even column
      // Original - 1 pixel from the "horizontal" estimate
      *Yptr = *Xptr - *(Xptr-lcount);
    }
  }
}
// Original - 3 pixels from the "cross" estimate
*Yptr = *Xptr - (*Xptr-lcount) + (*Xptr+lcount) +
       *(Xptr-W*lcount))/3;
} else {  // even column
// Original - 1 pixel from the "vertical" estimate
*Yptr = *Xptr - *(Xptr-W*lcount);
}

if(Weven) {
    // Original - 2 pixels from the "cross" estimate
*Yptr = *Xptr - (*Xptr-lcount) + *(Xptr-W*lcount))/2;
}

// adjust width, height and level pixel distance count for the next level
width = width/2 + 1 - Weven;
height = height/2 + 1 - Heven;
lcount *= 2;
}

/*
subsample inverse transform
X = original unchanged image (to be calculated)
width = width of original image (given)
height = height of original image (given)
Y = transformed image (given)
NOTE: Memory for X, Y, & Z allocated prior to calling this function
level = number of levels to apply the inverse transform, 0=None, 1=1, ...
*/
void invsub(unsigned char huge *X, int W, int H, int huge *Z, int level)
{
    int i, j, Weven, Heven, width, height, lcount;
    unsigned char huge *Xptr, huge *firstX;
    int huge *Y, huge *Yptr, huge *firstY, huge *Zptr;
    long numZ;

    numZ = (long)W * (long)H;
    Y = (int huge *) farmalloc((long)sizeof(int) * numZ);
    if(Y == NULL) return;
    Yptr = Y;
    Zptr = Z;
    while(numZ > 0) {

while(level > 0) {
    Icount = 1;
    width = W;
    height = H;
    if(width%2) Weven = 0;
    else Weven = 1;
    if(height%2) Heven = 0;
    else Heven = 1;
    for(i=1; i<level; i++) {
        Icount *= 2;
        width = width/2 + 1 - Weven;
        height = height/2 + 1 - Heven;
        if(width%2) Weven = 0;
        else Weven = 1;
        if(height%2) Heven = 0;
        else Heven = 1;
    }
    Xptr = X;
    Yptr = Y;
    // go through every pixel of original image
    // 1st calculating the horizontal and vertical estimates
    for(j=0; j<height-Heven; j++) {
        firstX = Xptr;
        firstY = Yptr;
        for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
            if(i%2) {  // odd column
                if(j%2) {  // odd row
                    // NOOP on 1st pass
                } else {  // even row
                    // Difference + 2 pixel "horizontal" estimate
                    *Xptr = *Yptr + (**(Yptr-lcount) + *(Yptr+lcount))/2;
                }
            } else {  // even column
                if(j%2) {  // odd row
                    // Difference + 2 pixel "vertical" estimate
                    *Xptr = *Yptr + (**(Yptr-W*lcount) + *(Yptr+W*lcount))/2;
                } else {  // even row
                    // NOOP on 1st pass
                }
            }
        }
    }
}
if(Weven) {
    if(i%2) {  // odd row
        // NOOP on 1st pass
        Xptr = *Yptr;
    } else {  // even row (unchanged sub-sample of original)
        *Xptr = *Yptr;
    }
}

if(Weven) {
    if(j%2) {  // odd row
        // NOOP on 1st pass
    } else {  // even row
        // Difference + 1 pixel from the "horizontal" estimate
        *Xptr = *Yptr + *(Yptr-lcount);
        *Yptr = *Xptr;
    }
}

if(Heven) {
    for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
        if(i%2) {  // odd column
            // NOOP on 1st pass
        } else {  // even column
            // Difference + 1 pixel from the "vertical" estimate
            *Xptr = *Yptr + *(Yptr-W*lcount);
            *Yptr = *Xptr;
        }
    }
}

if(Weven) {
    // NOOP on 1st pass
}

Xptr = firstX + W*lcount;
Yptr = firstY + W*lcount;

if(Heven) {
    for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
        if(i%2) {  // odd column
            // NOOP on 1st pass
        } else {  // even column
            // Difference + 1 pixel from the "vertical" estimate
            *Xptr = *Yptr + *(Yptr-W*lcount);
            *Yptr = *Xptr;
        }
    }
}

if(Weven) {
    // NOOP on 1st pass
}

Xptr = firstX + W*lcount;
Yptr = firstY + W*lcount;

// go through every pixel of original image
// 2nd calculating the "cross" estimates
for(j=0; j<height-Heven; j++) {
    firstX = Xptr;
    firstY = Yptr;
    for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
        if(i%2) {  // odd column
            if(j%2) {  // odd row
                // Difference + 4 pixel "cross" estimate
                *Xptr = *Yptr +
            }
        } else {  // even column
            if(j%2) {  // odd row
                // Difference + 4 pixel "cross" estimate
                *Xptr = *Yptr +
            }
        }
    }
}
(*(Yptr-lcount) + *(Yptr+lcount) +
*(Yptr-W*lcount) + *(Yptr+W*lcount))/4;
*Yptr = *Xptr;
} else { // even row
   // NOOP on 2nd pass
}
} else { // even column
if(j%2) { // odd row
   // NOOP on 2nd pass
} else { // even row (unchanged sub-sample of original)
   // NOOP on 2nd pass
}
}
if(Weven) {
if(j%2) { // odd row
   // Difference + 3 pixels from the "cross" estimate
   *Xptr = *Yptr + (*(Yptr-lcount) +
                   *(Yptr-W*lcount))/3;
   *Yptr = *Xptr;
} else { // even row
   // NOOP on 2nd pass
}
}
Xptr = firstX + W*lcount;
Yptr = firstY + W*lcount;
}
if(Weven) {
   for(i=0; i<width-Weven; i++, Xptr+=lcount, Yptr+=lcount) {
      if(i%2) { // odd column
         // Difference + 3 pixels from the "cross" estimate
         *Xptr = *Yptr + (*(Yptr-lcount) + *(Yptr+lcount) +
                           *(Yptr-W*lcount))/3;
         *Yptr = *Xptr;
      } else { // even column
         // NOOP on 2nd pass
      }
   }
}
if(Weven) {
   // Difference + 2 pixels from the "cross" estimate
   *Xptr = *Yptr + (*(Yptr-lcount) + *(Yptr-W*lcount))/2;
   *Yptr = *Xptr;
}
level--; 
}

DPCM Transform

/*
 XDPCM2.H

 DPCM transform using 1 pixel estimation
 */

#ifndef __XDPCM2_H
#define __XDPCM2_H

void xdpcm2(unsigned char huge *X, int W, int H, int huge *Y);
void invdpcm2(unsigned char huge *X, int W, int H, int huge *Y);

#endif

/*
 XDPCM2.CPP

 DPCM transform using 2 pixel estimation
 */

void xdpcm2(unsigned char huge *X, int W, int H, int huge *Y)
{
    int i, j;
    unsigned char huge *Xptr;
    int huge *Yptr;

    //Store the first pixel
    Xptr = X;
    Yptr = Y;
    *Yptr++ = *Xptr++;

    //Find the first row
    for(i=1; i<W; i++, Xptr++) *Yptr++ = *Xptr - *(Xptr-1);

    for(j=1; j<H; j++)
void invdpcm2(unsigned char huge *X, int W, int H, int huge *Y)
{
    int i, j;
    unsigned char huge *Xptr;
    int huge *Yptr;

    // Store the first pixel
    Xptr = X;
    Yptr = Y;
    *Xptr++ = *Yptr++;

    // Restore the first row
    for(i=1; i<W; i++, Xptr++) *Xptr = *Yptr++ + *(Xptr-1);

    for(j=1; j<H; j++) {
        // First column
        *Xptr = *Yptr++ + *(Xptr-W);
        Xptr++;

        // Middle pixels
        for(i=1; i<W; i++, Xptr++)
            *Xptr = *Yptr++ + (*Xptr-1) + *(Xptr-W) + 1)/2;
    }
}

Integer Wavelet Transform

/*

xintwav.h

Integer Wavelet transform & inverse transform header file

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
#ifndef _XINTWAV_H
#define _XINTWAV_H

#include <alloc.h>

/*
 *    INTWAV transform
 *    X = original unchanged image (given)
 *    width = width of original image (given)
 *    height = height of original image (given)
 *    Y = transformed image, (subsample included)
 *    level = number of levels to apply the transform, 0=none, 1=1, ...
 */
bool xintwav(unsigned char huge *X, int W, int H, int huge *Y, int level);

/*
 *    INTWAV inverse transform
 *    X = original unchanged image (to be calculated)
 *    width = width of original image (given)
 *    height = height of original image (given)
 *    Y = transformed image (given)
 *    level = number of levels to apply the inverse transform, 0=none, 1=1, ...
 */
bool invintwav(unsigned char huge *X, int W, int H, int huge *Y, int level);

#endif

/*
 xintwav.cpp
 */

#include "xintwav.h"
#include <stdio.h>

bool xintwav(unsigned char huge *X, int W, int H, int huge *Y, int level)
{
    int i, j, width, height, Width, Height, Wodd, Hodd;
    unsigned char huge *Xptr;
    int huge *A, huge *Aptr, huge *A2ptr;

int huge *Yptr, huge *Y2ptr;
long numcodes;

// determine width and height of transformed image
width = W;
height = H;
for(i=level; i>0; i--) {
    Wodd = width%2;
    Hodd = height%2;
    width = width/2 + Wodd;
    height = height/2 + Hodd;
}
for(i=level; i>0; i--) {
    width *= 2;
    height *= 2;
}

// allocate space for working copy of image A and transform image Y
numcodes = (long)width * (long)height;
A = (int huge *) farmalloc(sizeof(int) * numcodes);
if(A==NULL) return false;

// make working copy of original, extending last column/row to W & H of
// transformed image
    Aptr = A;
    Xptr = X;
    Yptr = Y;
    for(j=0; j<height; j++) {
        if(j>H) Xptr -= W;
        for(i=0; i<width; i++) {
            if(i<W) *Aptr++ = (int)*Xptr++;
            else *Aptr++ = (int)*(Xptr-1);
        }
    }
    Aptr = A;
    Yptr = Y;
    for(j=0; j<height; j++) {
        for(i=0; i<width; i++) {
            *Yptr++ = *Aptr++;
        }
    }

/* testing*/
FILE *fp;
fp = fopen("xiwtest0.dat", "w");

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Yptr = Y;
for(j=0; j<height; j++) {
    for(i=0; i<width; i++) {
        fprintf(fp, "%6d " , Yptr);
    }
    fprintf(fp, "\n" );
}
fclose(fp);

Width = width;
Height = height;
while(level-- > 0) {

    // horizontal component
    Aptr = A;
    Yptr = Y;
    for(j=0; j<height; j++) {
        Y2ptr = Yptr + width/2;
        for(i=0; i<width; i+=2, Aptr+=2) {
            *Yptr++ = *Aptr + *(Aptr+1);
            *Y2ptr++ = *Aptr - *(Aptr+1);
        }
        Yptr += (Width - width/2);
        Aptr += (Width - width);
    }

    // copy all Y values to A, (inefficient, but effective)
    Aptr = A;
    Yptr = Y;
    for(j=0; j<Height; j++)
        for(i=0; i<Width; i++)
            *Aptr++ = *Yptr++;

    // vertical component
    Aptr = A;
    Yptr = Y;
    Y2ptr = Y;
    for(j=0; j<height/2; j++) Y2ptr += Width;
    for(j=0; j<height; j+=2) {
        A2ptr = Aptr + Width;
        for(i=0; i<width; i++, Aptr++, A2ptr++) {
            *Yptr++ = *Aptr + *A2ptr;
            *Y2ptr++ = *Aptr - *A2ptr;
        }
    }
}
Aptr += (2*Width - width);
Yptr += (Width - width);
Y2ptr += (Width - width);
}

// copy all Y values to A, (inefficient, but effective)
Aptr = A;
Yptr = Y;
for(j=0; j<Height; j++)
  for(i=0; i<Width; i++)
    *Aptr++ = *Yptr++;

// set the width & height for next level o f transformation
width /= 2;
height /= 2;
}

/* testing */
fp = fopen("xiwtstN.dat", "w");
Yptr = Y;
for(j=0; j<Height; j++) {
  for(i=0; i<Width; i++) {
    fprintf(fp,"%6d ",*Yptr);
  }
  fprintf(fp,"\n");
}
fclose(fp);
*/

return true;
}

bool invintwav(unsigned char huge *X, int W, int H, int huge *Y, int level) {
  int i, j, width, height, Width, Height, Wodd, Hodd;
  unsigned char huge *Xptr;
  int huge *A, huge *Aptr, huge *A2ptr;
  int huge *B, huge *Bptr, huge *B2ptr;
  int huge *Yptr;
  long numcodes;

  // determine width and height of transformed image
  width = W;
  height = H;
  for(i=level; i>0; i--) {

Wodd = width%2;
Hodd = height%2;
width = width/2 + Wodd;
height = height/2 + Hodd;
}
for(; i<level; i++) {
    width *= 2;
    height *= 2;
}
Width = width;
Height = height;

// allocate space for working copies (A&B) of trans image transform image (B)
numcodes = (long)Width * (long)Height;
A = (int huge *) farmalloc(sizeof(int) * numcodes);
if(A==NULL) return false;
B = (int huge *) farmalloc(sizeof(int) * numcodes);
if(B==NULL) {
    farfree(A);
    return false;
}

// make working copies A & B of transformed image Y
Bptr = B;
Yptr = Y;
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Bptr++ = *Yptr++;

Aptr = A;
Bptr = B;
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Aptr++ = *Bptr++;

// start with lowest level
width = W;
height = H;
for(i=level; i>0; i--) {
    Wodd = width%2;
    Hodd = height%2;
    width = width/2 + Wodd;
    height = height/2 + Hodd;
}
width *= 2;
height *= 2;
while(level-- > 0) {

// horizontal component
Ap = A;
Bp = B;
for(j=0; j<height; j++) {
    B2p = Bp + width/2;
    for(i=0; i<width; i+=2, Ap+=2, Bp++, B2p++) {
        *Ap = (*Bp + *B2p)/2;
        *(Ap+1) = (*Bp - *B2p)/2;
    }
    Bp += (Width - width/2);
    Ap += (Width - width);
}

// copy all A values to B, (inefficient, but effective)
Bp = B;
Ap = A;
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Bp++ = *Ap++;

// vertical component
Ap = A;
Bp = B;
B2p = B;
for(j=0; j<height/2; j++) B2p += Width;
for(j=0; j<height/2; j+=2) {
    A2p = Ap + Width;
    for(i=0; i<width; i++, Bp++, B2p++) {
        // even row
        *Ap++ = (*Bp + *B2p)/2;
        // odd row
        *A2p++ = (*Bp - *B2p)/2;
    }
    // adjust pointers to next line
    Ap += (2*Width - width);
    Bp += (Width - width);
    B2p += (Width - width);
}

// copy all A values to B, (inefficient, but effective)
Bp = B;
Ap = A;
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Bptr++ = *Apctr++;

// set the width & height for next level of transformation
width *= 2;
height *= 2;
}

// Copy inverse transform back to the original
Apctr = A;
Xptr = X;
for(j=0; j<H; j++) {
    for(i=0; i<W; i++)
        *Xptr++ = *Apctr++;
    for(i=W; i<Width; i++) Xptr++;
}

return true;

Floating Point Wavelet Transform

/*

xfpwav.h

Floating Point Wavlet transform & inverse transform header file

*/

#ifndef __XFPWAV_H
#define __XFPWAV_H

#include<alloc.h>

/*

FPWAV transform
X = original unchanged image (given)
width = width of original image (given)
height = height of original image (given)
Y = transformed image,(subsample included)
level = number of levels to apply the transform, 0=none, 1=1, ...

*/
/*
bool xfpwav(unsigned char huge *X, int W, int H, int huge *Y, int level);
*/

/*
FPWAV inverse transform
X = original unchanged image (to be calculated)
width = width of original image (given)
height = height of original image (given)
Y = transformed image (given)
level = number of levels to apply the inverse transform, 0=none, 1=1, ...
*/
bool invfpwav(unsigned char huge *X, int W, int H, int huge *Y, int level, int numcolors);

#endif

#include "xfpwav.h"
#include <stdio.h>
#include <math.h>

bool xfpwav(unsigned char huge *X, int W, int H, int huge *Y, int level)
{
    int i, j, width, height, Width, Height, Wodd, Hodd;
    unsigned char huge *Xptr,
    int huge *A, huge *Aptr;
    int huge *Yptr, huge *Zptr;
    long numcodes;
    double a0, a1, a2;

    // calculate filter coefficients, with the best precision available
    a0 = (sqrt(2) + sqrt(14))/6.0;
    a1 = sqrt(2)*0.25;
    a2 = (sqrt(2) - sqrt(3.5))/6.0;

    // determine width and height of transformed image
    width = W;
    height = H;
    for(i=level; i>0; i--) {

Wodd = width%2;
Hodd = height%2;
width = width/2 + Wodd;
height = height/2 + Hodd;
}
for( i<level; i++) {
    width *= 2;
    height *= 2;
}

// allocate space for working copy of image A and transform image Y
numcodes = (long)width * (long)height;
A = (int huge *) farmalloc(sizeof(int) * numcodes);
if(A==NULL) return false;

// make working copy of original, extending last column/row to W & H of
// transformed image
    Aptr = A;
    Xptr = X;
    for(j=0; j<H; j++) {
        for(i=0; i<W; i++) *Aptr++ = *Xptr++;
        for( ; i<width; i++, Aptr++) *Aptr = *(Aptr-1);
    }
    for(; j<height; j++) {
        for(i=0; i<width; i++, Aptr++) *Aptr = *(Aptr-width);
    }
    Aptr = A;
    Yptr = Y;
    for(j=0; j<height; j++) {
        for(i=0; i<width; i++) {
            *Yptr++ = *Aptr++;
        }
    }

/* testing
FILE *fp;
fp = fopen("xfpwtst0.dat", "w");
Yptr = Y;
for(j=0; j<height; j++) {
    for(i=0; i<width; i++) {
        fprintf(fp,"%6d ",*Yptr);
    }
    fprintf(fp,"\n");
}
close(fp);
/*

Width = width;
Height = height;
while(level-- > 0) {

// horizontal component
Apotr = A;
Yptr = Y;
for(j=0; j<height; j++) {
    Zptr = Yptr + width/2;
    for(i=0; i<width; i++, Apotr) {
        if(i%2) {  // odd column (Z's)
            if(i==1) {
                *Zptr++ = (int)((double)*(Apotr+2) * a2 -
                           (double)*(Apotr-1) * a1 +
                           (double)*Apotr * a0 -
                           (double)*(Apotr+1) * a1 +
                           (double)*(Apotr+2) * a2 + 0.5);
            } else if(i==width-2) {
                *Zptr++ = (int)((double)*(Apotr-2) * a2 -
                           (double)*(Apotr-1) * a1 +
                           (double)*Apotr * a0 -
                           (double)*(Apotr+1) * a1 +
                           (double)*(Apotr-2) * a2 + 0.5);
            } else if(i==width-1) {
                *Zptr++ = (int)((double)*(Apotr-2) * a2 -
                           (double)*(Apotr-1) * a1 +
                           (double)*Apotr * a0 -
                           (double)*(Apotr+1) * a1 +
                           (double)*(Apotr-2) * a2 + 0.5);
            } else {
                *Zptr++ = (int)((double)*(Apotr-2) * a2 -
                           (double)*(Apotr-1) * a1 +
                           (double)*Apotr * a0 -
                           (double)*(Apotr+1) * a1 +
                           (double)*(Apotr+2) * a2 + 0.5);
            }
        } else {  // even column (Y's)
            if(i==0) {
                *Yptr++ = (int)((double)*(Apotr+2) * a2 +
                           (double)*(Apotr+1) * a1 +
                           (double)*Apotr * a0 +
                           (double)*(Apotr+1) * a1 +
                           (double)*(Apotr+2) * a2 + 0.5);
            }
        }
    }
}
}
if (i==width-2) {
    *Yptr++ = (int)((double)*(Aptr-2) * a2 +
    (double)*(Aptr-1) * a1 +
    (double)*Aptr * a0 +
    (double)*(Aptr+1) * a1 +
    (double)*(Aptr-2) * a2 + 0.5);
} else if(i==width-1) {
    *Yptr++ = (int)((double)*(Aptr-2) * a2 +
    (double)*(Aptr-1) * a1 +
    (double)*Aptr * a0 +
    (double)*(Aptr-1) * a1 +
    (double)*(Aptr-2) * a2 + 0.5);
} else {
    *Yptr++ = (int)((double)*(Aptr-2) * a2 +
    (double)*(Aptr-1) * a1 +
    (double)*Aptr * a0 +
    (double)*(Aptr-1) * a1 +
    (double)*(Aptr+2) * a2 + 0.5);
}
}
Yptr += (Width - width/2);
Aptr += (Width - width);

// copy all Y values to A, (inefficient, but effective)
Aptr = A;
Yptr = Y;
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Aptr++ = *Yptr++;

// vertical component
Aptr = A;
Yptr = Y;
Zptr = Y;
for(j=0; j<height/2; j++) Zptr += Width;
for(j=0; j<height; j++) {
    for(i=0; i<width; i++, Aptr++) {
        if(j%2) { // odd row(YZ's)
            if(i==1) {
                *Zptr++ = (int)((double)*(Aptr+2*Width) * a2 -
                (double)*(Aptr-Width) * a1 +
                (double)*Aptr * a0 -
                (double)*(Aptr+Width) * a1 +
            }
        }
    }
}
(double)*(Aptr+2*Width) * a2 + 0.5);
} else if (j==height-2) {
    *Zptr++ = (int)((double)*(Aptr-2*Width) * a2 -
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 -
(double)*(Aptr+Width) * a1 +
(double)*(Aptr-2*Width) * a2 + 0.5);
} else if (j==height-1) {
    *Zptr++ = (int)((double)*(Aptr-2*Width) * a2 -
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 -
(double)*(Aptr+Width) * a1 +
(double)*(Aptr-2*Width) * a2 + 0.5);
} else {
    *Zptr++ = (int)((double)*(Aptr-2*Width) * a2 -
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 -
(double)*(Aptr+Width) * a1 +
(double)*(Aptr+2*Width) * a2 + 0.5);
}
} else { // even row (YY's)
    if (j==0) {
        *Yptr++ = (int)((double)*(Aptr+2*Width) * a2 +
(double)*(Aptr+Width) * a1 +
(double)*Aptr * a0 +
(double)*(Aptr+Width) * a1 +
(double)*(Aptr+2*Width) * a2 + 0.5);
    } else if (j==height-2) {
        *Yptr++ = (int)((double)*(Aptr-2*Width) * a2 +
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 +
(double)*(Aptr+Width) * a1 +
(double)*(Aptr+2*Width) * a2 + 0.5);
    } else if (j==height-1) {
        *Yptr++ = (int)((double)*(Aptr-2*Width) * a2 +
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 +
(double)*(Aptr+Width) * a1 +
(double)*(Aptr+2*Width) * a2 + 0.5);
    } else {
        *Yptr++ = (int)((double)*(Aptr+2*Width) * a2 +
(double)*(Aptr-Width) * a1 +
(double)*Aptr * a0 +
(double)*(Aptr+Width) * a1 +
(double)*(Aptr+2*Width) * a2 + 0.5);
    }
}
/* testing */
fp = fopen("xfpwstN.dat", "w");
Yptr = Y;
for(j=0; j<Height; j++) {
    for(i=0; i<Width; i++) {
        fprintf(fp, "%6d ", *Yptr);
    }
    fprintf(fp, "\n");
}
fclose(fp);
*

return true;
}

bool invfpwav(unsigned char huge *X, int W, int H, int huge *Y, int level, int numcolors) {
    int i, j, width, height, Width, Height, Wodd, Hodd;
    unsigned char huge *Xptr;
    int huge *A, huge *Aptr, huge *A2ptr;
    int huge *B, huge *Bptr, huge *B2ptr;
    int huge *Yptr;
    long numcodes;
    double a0, a1, a2;
// calculate filter coefficients, with the best precision available
a0 = (sqrt(2) + sqrt(14))/6.0;
a1 = sqrt(2)*0.25;
a2 = (sqrt(2) - sqrt(3.5))/6.0;

// determine width and height of transformed image
width = W;
height = H;
for(i=level; i>0; i--) {
    Wodd = width%2;
    Hodd = height%2;
    width = width/2 + Wodd;
    height = height/2 + Hodd;
}
for(; i<level; i++) {
    width *= 2;
    height *= 2;
}
Width = width;
Height = height;

// allocate space for working copy of image A and transform image Y
numcodes = (long)Width * (long)Height;
A = (int huge *) farmalloc(sizeof(int) * numcodes);
if(A==NULL) return false;
B = (int huge *) farmalloc(sizeof(int) * numcodes);
if(B==NULL) {
    farfree(A);
    return false;
}

/* testing
fp = fopen("xfpwinvN.dat", "w");
Yptr = Y;
for(j=0; j<Height; j++) {
    for(i=0; i<Width; i++) {
        fprintf(fp,"%6d ",*Yptr++);
    }
    fprintf(fp,"
");
} fclose(fp);
*/

// make working copies A & B of transformed image Y
Bptr = B;
Yptr = Y;
for(j=0; j<Height; j++)
  for(i=0; i<Width; i++)
    *Bptr++ = *Yptr++;

Aptr = A;
Bptr = B;
for(j=0; j<Height; j++)
  for(i=0; i<Width; i++)
    *Aptr++ = *Bptr++;

// start with lowest level
width = W;
height = H;
for(i=level; i>0; i--) {
  Wodd = width%2;
  Hodd = height%2;
  width = width/2 + Wodd;
  height = height/2 + Hodd;
}
width *= 2;
height *= 2;
while(level-- > 0) {

// horizontal component
Aptr = A;
Bptr = B;
for(j=0; j<height; j++) {
  B2ptr = Bptr + width/2;
  for(i=0; i<width; i+=2, Bptr++, B2ptr++) {
    if(i==0) {
      *Aptr++ = (int)((double)*(Bptr+1) * a2 -
        (double)*B2ptr * a1 +
        (double)*Bptr * a0 -
        (double)*B2ptr * a1 +
        (double)*(Bptr+1) * a2 + 0.5);
      *Aptr++ = (int)((double)*(B2ptr+1) * a2 +
        (double)*Bptr * a1 +
        (double)*B2ptr * a0 +
        (double)*(Bptr+1) * a1 +
        (double)*(B2ptr+1) * a2 + 0.5);
    } else if(i==width-2) {
      *Aptr++ = (int)((double)*(Bptr-1) * a2 -
        (double)*Bptr * a1 +
        (double)*B2ptr * a0 +
        (double)*(Bptr+1) * a1 +
        (double)*(B2ptr+1) * a2 + 0.5);
    } else if(i==width-1) {
      *Aptr++ = (int)((double)*(Bptr) * a2 -
        (double)*Bptr * a1 +
        (double)*B2ptr * a0 +
        (double)*(Bptr+1) * a1 +
        (double)*(B2ptr+1) * a2 + 0.5);
    } else {
      *Aptr++ = (int)((double)*(Bptr+1) * a2 -
        (double)*Bptr * a1 +
        (double)*B2ptr * a0 +
        (double)*(Bptr+1) * a1 +
        (double)*(B2ptr+1) * a2 + 0.5);
    }
  }
}

// vertical component
Aptr = A;
Bptr = B;
for(i=0; i<width; i++)
  for(j=0; j<height; j++)
    *Aptr++ = *Bptr++;

// end with lowest level
width = W;
height = H;
for(i=level; i>0; i--) {
  Wodd = width%2;
  Hodd = height%2;
  width = width/2 + Wodd;
  height = height/2 + Hodd;
}
width *= 2;
height *= 2;
(double)*(B2ptr-l) * a1 +
(doubIe)*Bptr * a0 -
(dounde)*B2ptr * a0 +
(doubIe)*(Bptr-1) * a2 + 0.5);
*Aptr++ = (int)((doubIe)*(B2ptr-1) * a2 +
(double)*Bptr * al +
(double)*B2ptr * a0 +
(double)*Bptr * a0 +
(double)*(B2ptr-1) * a2 + 0.5);

} else {

    *AptrH- = (intX((doubIe)*(Bptr-l) ♦ a2 -
    (doublde)*(B2ptr-l) ♦ a1 +
    (doulde)*Bptr ♦ a0 -
    (doulde)*B2ptr ♦ a0 +
    (doulde)*(Bptr+l) ♦ a2 + 0.5);
    *Aptr++ = (int)((doubIe)*(B2ptr-l) ♦ a2 +
    (doubde)*Bptr ♦ a1 +
    (doulde)*B2ptr ♦ a0 +
    (doulde)*(Bptr+l) ♦ a1 +
    (doulde)*(B2ptr+l) ♦ a2 + 0.5);

}

Bptr += (Width - width/2);
Aptr += (Width - width);

// copy all A values to B, (inefficient, but effective)
Bptr = B;
Aptr = A;
for(j=0; j<Height; i++)
    for(i=0; i<Width; i++)
        *Bptr++ = *Aptr++;

// vertical component
Aptr = A;
Bptr = B;
B2ptr = B;
for(j=0; j<height/2; j++) B2ptr += Width;
for(j=0; j<height; j++) { // even row
    A2ptr = Aptr + Width;
    for(i=0; i<width; i++, Bptr++, B2ptr++) {
        if(j==0) {
            *Aptr++ = (int)((double)*(Bptr+Width) ♦ a2 -
            (dovlde)*B2ptr ♦ a1 +

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
(double)*Bptr * a0 -
(double)*B2ptr * a1 +
(double)*(Bptr+Width) * a2 + 0.5);

// odd row
*Aptr++ = (int)((double)*(B2ptr+Width) * a2 +
(double)*Bptr * a1 +
(double)*B2ptr * a0 +
(double)*(Bptr+Width) * a1 +
(double)*(B2ptr+Width) * a2 + 0.5);

} else if(j==height-2) {

// even row
*Aptr++ = (int)((double)*(Bptr-Width) * a2 -
(double)*Bptr * a0 -
(double)*B2ptr * a1 +
(double)*(Bptr-Width) * a1 +
(double)*(Bptr+Width) * a2 + 0.5);

// odd row
*A2ptr++ = (int)((double)*(B2ptr-Width) * a2 +
(double)*Bptr * a0 +
(double)*B2ptr * a1 +
(double)*(Bptr-Width) * a1 +
(double)*B2ptr * a2 + 0.5);

} else {

// even row
*Aptr++ = (int)((double)*(Bptr-Width) * a2 -
(double)*(B2ptr-Width) * a1 +
(double)*Bptr * a0 -
(double)*B2ptr * a1 +
(double)*(Bptr+Width) * a2 + 0.5);

// odd row
*A2ptr++ = (int)((double)*(B2ptr-Width) * a2 +
(double)*Bptr * a0 +
(double)*B2ptr * a1 +
(double)*(Bptr+Width) * a1 +
(double)*(B2ptr+Width) * a2 + 0.5);

}

Aptr += (2*Width - width);
Bptr += (Width - width);
B2ptr += (Width - width);

} else if(j==height-2) {
// even row
*Aptr++ = (int)((double)*(Bptr-Width) * a2 -
(double)*(B2ptr-Width) * a1 +
(double)*Bptr * a0 -
(double)*B2ptr * a1 +
(double)*(Bptr+Width) * a2 + 0.5);

// odd row
*A2ptr++ = (int)((double)*(B2ptr-Width) * a2 +
(double)*Bptr * a0 +
(double)*B2ptr * a1 +
(double)*(Bptr+Width) * a1 +
(double)*(B2ptr+Width) * a2 + 0.5);

}

// copy all A values to B, (inefficient, but effective)
Bptr = B;
Aptr = A;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
for(j=0; j<Height; j++)
    for(i=0; i<Width; i++)
        *Bptr++ = *Aptr++;

// set the width & height for next level of transformation
width *= 2;
height *= 2;
}

// Copy inverse transform back to the original
Aptr = A;
Xptr = X;
for(j=0; j<Height; j++) {
    for(i=0; i<Width; i++) {
        if(*Aptr < 0) *Xptr++ = 0;
        else if(*Aptr >= numcolors) *Xptr++ = (unsigned char)numcolors-1;
        else *Xptr++ = *Aptr;
        Aptr++;
    }
    for(; i<Width; i++) Aptr++;
}

/* testing
FILE *fp;
fp = fopen("xfpwinv0.dat", "w");
Xptr = X;
for(j=0; j<Height; j++) {
    for(i=0; i<Width; i++) {
        fprintf(fp,"%6d ,, *Xptr++);
    }
    fprintf(fp,"\n");
}
fclose(fp);
*/

return true;
APPENDIX II

SAMPLE IMAGES

All images are 8 bit gray-scale TIFF images. (Tagged Image File Format)

"frame1.tif"
“frame3.tif”
"frame4.tif"
“frame5.tif"
REFERENCES


