Lossless Image Compression Based on Two-Way Smaller Difference

*Chin-Chen Chang, *Chi-Shiang Chan and **Ju-Yuan Hsiao

**Department of Computer Science and Information Engineering*
National Chung Cheng University
E-mail: {ccc, cch}@cs.ccu.edu.tw

***Department of Information Management*
National Changhua University of Education
E-mail: hsiaojy@cc.ncue.edu.tw

Abstract

In this paper, we shall propose a lossless image compression method, in which we compress an image by saving the differences between the encoding pixels and their adjacent pixels. To deal with a pixel, we refer to not only its up pixel but also its left pixel and replace the value of the pixel itself with the value of difference. The experimental results show that the method gives quite an impressive performance. Since the values of most adjacent pixels in an image are closed, we can use this simple, effective way to compress the image.

1. INTRODUCTION

Data compression plays an important role in computer storage and transmission. The purpose of data compression is that we can reduce the size of data to save storage and reduce time for transmission. The same thing goes for image compression; namely, we can save storage by using image compression. There are two main domains that have been raised for research in image compression: One is lossy image compression [1,5] and the other is lossless image compression [7,2,6,4]. For example, a famous scheme has been created by modifying the data compression technique called Lempel-Ziv-Welch (LZW) [7] to do lossless image compression; likewise, in [6], Ng and Cheng modified a well-established text compression method, Burrows and Wheeler Transformation (BWT) [2,4], is also for image compression. Our method too is focused on lossless image compression.

Basically, we compress an image following the agreed-upon assumption that the values of most adjacent pixels in an image are similar if not identical. We first check the difference between the values of the up pixel and the left pixel of the target pixel. If the difference is greater than a given threshold, we set a bit to point out which one to refer to. If the difference falls under the threshold, we can replace the target with the value of difference. The experimental results show that our method gives quite an impressive performance.

The rest of this paper is organized as follows. We shall describe the encoding method as well as decoding method for one-level lossless image compression in Section 2.1 and that for two-level lossless image compression in Section 2.2. After that, we shall portray our method in Section 2.3. In Section 3, the experimental results will show how our method performs. Finally, the conclusions will be presented in Section 4.

2. The Proposed Scheme

2.1 The encoding and decoding method for one-level lossless image compression

In this section, we shall introduce the one-level lossless image compression method. Basically, the images are encoded row by row. We start by defining some operations we need to do. First of all, there are two buffers, called *subtraction_buffer* and *real_pixels_buffer*, with both of their lengths are equal to that of the row. To begin encoding one row of the image, we record the real value of the first pixel in the first element of *subtraction_buffer*. Then we get the value via subtracting the real value in previous position from that of its position and record the result into *subtraction_buffer*. The range of value in *subtraction_buffer* can be from –255 to 255, which makes it hard for us to benefit from compression; therefore, we set the number of bits, called *bit_number,* as the threshold. It means that we use this threshold to check whether or not the value in *subtraction_buffer* is in the range so that the size of the target pixel value is $\boldsymbol{x}$ bits, where $\boldsymbol{x}$ is the value of *bit_number*. Here we define the variable *bit_number_range* as the range within which *bit_number* can be expressed by way of two's complement. If the value of an element in *subtraction_buffer* is in the range, then we keep it in *subtraction_buffer*; otherwise, we have to set a special symbol in the position in order to indicate that the value of this position must be gotten from another buffer. Once we finish setting the special symbol in the position, we record the real value of the pixel in the relative position of *real_pixels_buffer*. After doing so, we have set the element's value of *subtraction_buffer* in the range of *bit_number_range*. It means that we can just spend *bit_number* bits to represent every element of *subtraction_buffer* by way of two's complement, and therefore we can get the code sequences from encoding *subtraction_buffer*. We now continue to encode *real_pixels_buffer*. If the elements of *real_pixels_buffer* are not blank, then we append them to the code sequences with eight bits. At the end of appending, we can get the code sequences with enough information to recover this row. We do this row by row and append encoded data to code sequences until we reach the end of the images.

If we want to extract the image from the code sequences, we must also recover it row by row. Here is how to decode the first row, and the other rows will be processed exactly the same way. First of all, we check the value of *bit_number* so that we know how many bits we need to extract at a time. Each time, we extract as many as *bit_number* bits and use two's complement to represent the value. We insert the value into an element of *subtraction_buffer* at a time and do it over and over again until the value is inserted into all the elements of *subtraction_buffer*. Now that we have filled out all the elements in *subtraction_buffer*, we can scan it to recover the real pixel values. Please note that all the values of the elements in *subtraction_buffer* are either the result of a subtraction or the value of a special symbol. We must scan all the elements in *subtraction_buffer* again. If the value of a certain element in *subtraction_buffer* is the value of a special symbol, we extract next eight bits from the code sequences. We directly insert eight bits into the relative position of the element in *subtraction_buffer* as a real pixel. On the other hand, if the value of a certain element is not the value of a special symbol, then we add the value and its previous value in *subtraction_buffer* up and insert the result into the relative position of the element in *subtraction_buffer* as a real pixel value.

2.2 The encoding and decoding method for two-level lossless image compression

We shall have a close look at the two-level lossless image compression in this section. Again, it is a row-by-row design. The method uses not only the two buffers mentioned earlier, namely *subtraction_buffer* and *real_pixels_buffer,* but also takes in a new buffer called *second_level_buffer*, which is inserted in the middle of the two earlier buffers. In Section 2.1, we use *bit_number* to check whether or not the value in *subtraction_buffer* is in the range of *bit_number_range*. If the value of an element in *subtraction_buffer* is in the range, then we keep it in *subtraction_buffer*; otherwise, we will set a special symbol in the position in order to indicate that the value of this position must be gotten from *real_pixels_buffer*. In the two-level lossless image compression, we do not record real pixels in the buffer of *real_pixels_buffer* immediately. Before recording the real pixels, we check whether or not the value in *subtraction_buffer* is in the range of 2* *bit_number_range*. If the value is not in the range, we record the special symbol in the relative position of *second_level_buffer* and record the real pixel value in the relative position of *real_pixels_buffer*. If the value is in the range, we do something to let the value fall into the range of *bit_number_range*. Because the values are in the range from *bit_number_range* to 2**bit_number_range*, we can limit the value in the range of *bit_number_range* by testing to see whether the value is. If it is a positive value, we can subtract the maximum value of *bit_number_range* from it and save the result in the relative position of *second_level_buffer*. If it is a negative value, we can add the maximum value of *bit_number_range* to it and save the result in the relative position of *second_level_buffer*. According to the description, we can also just spend *bit_number* bits to represent every element in *second_level_buffer* by way of two's complement, and therefore we append the relative position of *second_level_buffer* and *real_pixels_buffer* at the same time. If the same positions of *second_level_buffer* or *real_pixels_buffer* is not blank, we append them to the code sequences.

When we decode, we do the same work to check the value of *bit_number* so that we know how many bits we need to extract at a time. Each time, we extract *bit_number* bits and represent the value with two's complement. We insert the value into the element of *subtraction_buffer* and repeat that again and again until the value is inserted into all the elements of *subtraction_buffer*. Since we have filled out all the elements of *subtraction_buffer*, we can scan it pixel by pixel. If the value in *subtraction_buffer* is not the value of a special symbol, we add the value and its previous value in *subtraction_buffer* up and insert the result into the relative position of the element in *subtraction_buffer* as a real pixel value. If the value in *subtraction_buffer* is the value of a special symbol, we will extract next *bit_number* bits from the code sequences. If the value, just extracted from the code sequences, is also the value of a special symbol, we directly extract eight bits and insert them into the relative position of the element in *subtraction_buffer* as a real pixel value. If the value, just extracted from the code sequence, is still not the value of a special symbol, we test and see whether the value is positive or negative in order to decide to add or subtract the maximum value of *bit_number_range* to or from it. After doing that, we add up the value and its previous value in *subtraction_buffer* and insert the result into the relative position of the element in *subtraction_buffer* as a real pixel. When we have inserted all the real pixel values of the first row, we have already recovered the first row of the original image.

According to the our intuition, we can understand that, in lossless image compression, some images are suitable for row-by-row encoding while others are more suitable for column-by-column encoding. It is because images pixels tend to approximate to their neighboring pixels, sometimes to their left and sometimes to their top, and consequently some images are more properly processed row-by-row and others column-by-column. That is the reason why we modify the method and refer to the neighboring pixel of the target pixel in an image sometimes as its left pixel and sometimes as its up pixel. Let's further develop this notion in section 2.3.

2.3 The proposed method

In this section, we shall offer a detailed description of our proposed method. For us, there are two phases to compress images. The major job of the first phase is to decide the direction for encoding, and the second phase is aimed at encoding the images. We set two variables as counters, called *row_counter* and *column_counter*. The first phase starts by testing whether we can benefit more from row-by-row encoding or column-by-column encoding. The method scans all the pixels in the image and check if the differences are in the range of *bit_number_range.* If one difference is confirmed to the situation for its left-pixel checking, it means that this particular pixel suits row-by-row encoding, and so we add one to *row_counter*. On the other hand, if one difference is confirmed to the situation for its up-pixel checking, it means that this particular pixel suits column-by-column encoding, and so we add one to *column_counter*. After scanning the whole image, we compare the final value of *row_counter* with that of *column_counter* to decide the encoding direction and record it in the first bit of the code sequence. Now that we have already decided the direction and finished first phase, we continue to do the second phase, that is, the encoding phase. In the second phase, first of all, we allocate another new array, called *dir_buffer*, whose size is equal to the row length of the original image. Then we follow the default direction decided in the first phase to encode the image. When we begin to encode a pixel, we do some checking as follows:

1. When we encode a pixel, we check if the difference between its left pixel and its up pixel is in the range of *bit_number_range*. If the difference falls in the range, then we encode the pixel following the steps in Section 2.2 and use the default direction as the encoding direction. If it does not fall in the range, it means that they have a great difference between the values of the left pixel and the up pixel. The value of the target pixel can be either closer to its left pixel or to its up pixel.

2. If the difference between the left pixel value and the up pixel value of the pixel being encoded is not in the range of *bit_number_range*, then we check to see whether the smaller difference, which could be either between the target pixel value and its left pixel value or between the target pixel value and its up pixel value, is in the range of 2**bit_number_rang*. If yes, we will write 1 or 0 in the relative position of *dir_buffer* to point out which pixel we will refer to and use the method in Section 2.2 for encoding . If it disagrees with the condition, then we write a special symbol in the relative position of both *subtraction_buffer* and *second_level_buffer* and write the real pixel value in the relative position of *real_pixels_buffer*..

After inserting, we spend *bit_number* bits to represent every element of *subtraction_buffer* with two's complement, and thus we get the code sequences from encoding *subtraction_buffer*. We now continue to encode *second_level_buffer, real_pixels_buffer* and *dir_buffer*. We append the relative position of *second_level_buffer, real_pixels_buffer* and *dir_buffer* at the same time. If the same position element of *second_level_buffer* or *real_pixels_buffer* or *dir_buffer* is not blank, then we append them to code sequences. When the appending is done, we can get a string of code sequences.

If we want to decode the code sequence that is encoded above, we have to first extract the first bit in order to know the default direction. Then we get to check the value of *bit_number* and extract the bits time after time. After extracting the bits, we represent the value of those bits with two's complement and insert the value into the element of *subtraction_buffer*. We will repeat this process again to fill out all the elements of *subtraction_buffer*. After we finish inserting values, we can get to recover their real pixel values one by one. We check the difference between the left pixel value and the up pixel value of every target pixel in order to know whether or not the difference is in the range of *bit_number_range*.

1. If the difference is in the range of *bit_number_range*, we decode it following the steps in Section 2.2.

2. If the difference goes beyond the range of *bit_number_range* and not all the values of the relative position of *subtraction_buffer* and *second_level_buffer* are special symbols, we also use the decoding way in Section 2.2, but with a slight difference. We follow the steps in Section 2.2 till only as far as computing the difference. Then we extract the next bit in order to decide which pixels to refer to. After we get the reference pixel, we can decode and recover the original pixel.

3. If the difference is not in the range of *bit_number_range* and the relative position of both *subtraction_buffer* and *second_level_buffer* are special symbols, we extract the next eight bits and compute their value as the real pixel.

After doing that, we have recovered one pixel, and we can follow the same route to decode all the rest of the image.

3. Experimental Results

We use four 256-level gray scale images - Lena, Plane, Peppers and Tiffany – as test images. We also set the range of *bit_number* to be three and four bits in order to get the best performance. The meaning of "compression ratio" in all tables below is that we divide original image size by compressed image size. In our experimental results, the best performance is given when the *bit_number* is equal to four. Table 1 illustrates the results of our proposed method and the other methods. The method of GIF standard in Table 1 is modified from the method of Lempel-Ziv-Welch [7] for lossless image compression and the method of PKZIP is a standard for data compression. Burrows and Wheeler formed their BWT [2,6] by using two stages, transform and sorting, to compress text data. Ng and Cheng introduced sub-block interchange (SBI)[6] to overcome the drawback of BWT so that it can suit lossless image compress. We show the result of the Ng and Cheng's method in Table 2. The compression rate for Chuang and Lin's method [3] is represented in Table 3.

Table 1: Compression ratio of different methods.

	GIF	PKZIP	BWT	Proposed Method (Bit_number =3)	Proposed Method (Bit_number =4)
Lena	0.99	1.18	1.43	1.60	1.64
Plane	1.29	1.47	1.68	1.69	1.66
Peppers	0.97	1.13	1.44	1.66	1.70
Tiffany	1.15	1.34	1.51	1.68	1.71

Table 2: Compression ratio of Ng and Cheng's method.

	Lena	Peppers	Tiffany
Ng and Cheng's Method	1.45	1.44	1.52

Table 3: Compression ratio of Chuang and Lin's method.

	Lenna	Plane
Chuang and Lin's Method	1.51	1.60

4. Conclusions

A good lossless image compression method should take into consideration the characteristics of particular images so that we can benefit from compression. Both row-by-row encoding and column-by-column encoding refer to the values of pixels in one fixed direction and thus are not able to adjust themselves according to the characteristics of the images. Our method tries to combine the two methods into one. In our method, we refer to not only the left pixels but also the up pixels. According to the experimental results, we can say that our method gives quite an impressive performance.

5. REFERENCES

[1] O. K. Al-Shaykh, and R. M. Mersereau, "Lossy Compression of Noisy Images," *IEEE Transactions on Image Processing*, vol. 7, no. 12, pp 1641-1652, 1998.

[2] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," SRC Research Report 124,Digital Systems Research Center, Palo, Alto, May 1994.

[3] T. J. Chuang and J. C. Lin, "New Approach to Image Encryption," *Journal of Electronics Imaging,* vol. 7, no. 2,pp 350-356, 1998.

[4] P. Fenwick, "The burrows-Wheeler Transform for Block Sorting Text Compression: Principle and Improvement," *Computer Journal* , vol.39, no. 9, pp.731-740, 1996.

[5] S. W. Hong, and P. Bao, "Hybrid Image Compression Model Based On Subband Coding and Edge-Preserving Reqularisation," *IEE Proceedings on Vision, Image and Signal Processing*, vol. 147, pp.16 –22, 2000.

[6] K. S. Ng, and L. M. Cheng, "Sub-block Interchange for Lossless Image Compression ," *IEEE Transactions on Consumer Electronics*, vol. 45, no. 1, pp 236 – 242, 1999.

[7] I. Witten, A. Moffat and T. C. Bell, "Managing Gigabytes – Compressing and Indexing Documents and Images," Van Nostrand Reinhold, 1994.