Redistribution of the Compressed Data Between Modified DEFLATE-Blocks in the Image Compression Process Without Lossless

Andrii Bomba[1], Alexander Shportko[2] and Veronika Postolatii[3]

[1] *National University of Water and Environmental Engineering, 11, Soborna Str, 33028, Rivne, Ukraine*
[2] *Academician Stepan Demianchuk International University of Economics and Humanities, 4, Acad. S. Demianchuk Str, 33000, Rivne, Ukraine*
[3] *National University "Lviv Polytechnic", 12, St. Bandera Str, 79000, Lviv, Ukraine*

Abstract
Reasonable expediency of each stage of lossless image compression. The advantages and disadvantages of the DEFLATE data compression format are described. The feasibility of using arithmetic coding instead of Huffman compression in modifications of this format is argued. The principles of redistribution of compressed data between DEFLATE- blocks in order to increase their homogeneity to reduce data compression coefficients are presented and substantiated. The software implementation of this redistribution of compressed data between DEFLATE blocks in C++ language is given. On commonly known test the ACT set shows that application proposed redistribution of coded data between modified DEFLATE blocks in the process of lossless progressive hierarchical compression enables reduce the total size of its compressed images by 3 KB and speeds up decoding by an average of 2.14%, although it slows down encoding by 6.48 %.

Keywords
Image compression lossless, DEFLATE compressed data format, LZ77 dictionary compression algorithm.

1. Introduction

Today, files transmitted by communication channels most often contain data of one of four main types: texts, images, video or sound. And if 4 KB is enough to store one page of unformatted text, then a photo-realistic raster image of 10×15 cm size without compression can take several MB, song recordings - tens of MB, and movie recordings - several GB. Fortunately, most of these files have a high level of redundancy. It is the reduction of redundancy that allows you to compress files and thereby increase the speed of information exchange over the network and reduce the amount of disk space used.

Most of the well-known algorithms that allow you to compress images dozens of times (JPEG [1][2], fractal and wavelet transforms [3]) lead to slight loss of image quality. Such losses are invisible for photorealistic high-resolution images, but affect the quality of low-resolution images with fragments of business graphics or discrete-tone transitions. In addition, there are classes of images for which distortions are unacceptable (for example, X-rays). Algorithms for image compression without loss [4], although they compress images much weaker, do not lead to deterioration of their quality. Currently, lossy image compression algorithms are developing most dynamically, although lossless image compression algorithms are also gradually increasing their compression rates. Therefore, in the era of information civilization, when most information is stored electronically, the problem of lossless image compression by reducing redundancy remains relevant now and will be relevant in the future.

COLINS-2024: 8th International Conference on Computational Linguistics and Intelligent Systems, April 12–13, 2024, Lviv, Ukraine
abomba@ukr.net (A. Bomba); ITShportko@gmail.com (A. Shportko); VeronikaShportko@gmail.com (V. Postolatii)
0000-0001-5528-4192 (A. Bomba); 0000-0002-4013-3057 (A. Shportko); 0000-0002-9460-0781 (V. Postolatii)

2. Related works. Stages lossless compression of images

As it is known, any data compression is possible precisely due to the reduction or elimination of redundancies [5]. Three main types of redundancies are distinguished in images [6]: visual (consisting in the presence of information that is not perceived by the human visual system), inter-element or spatial (manifested in the correlation of brightness of adjacent pixels or components of the color model) and code (revealed when using codes of the same length for items with different probabilities). The more types of redundancies of each type are processed in a graphic format, the more effective the compression is. In the process of lossless compression, no information is lost, so the first type of redundancy is not reduced.

To reduce redundancies of the second and third types in archivers and graphic formats lossless compression of images most often occurs in a maximum of four stages:

1. At the first stage, context-dependent coding reduces redundancies between the same fragments or fragments with the same structure (reduces inter-element redundancy, can be also used before the fourth stage). For example, the popular context-dependent algorithm LZ77 [7] is based on the replacement in the output stream of the sequence of consecutive literals of the buffer with a reference to a similar pre-encoded sequence of dictionary literals in the form of a pair of numbers <*length*, *offset from the end of the dictionary*> in the process of coding. If there is no similar sequence of literals in the dictionary longer than two literals, the first literal of the buffer is transferred to the output stream without changes. After that, the encoded literals are transferred from the beginning of the buffer to the end of the dictionary and the encoding continues similarly until the end of the literals of the input stream. An example of the step-by-step results of the LZ77 algorithm scheme formation for flow *3, 4, 6, 3, 4, 6, 3, 2, 6, 3, 4, 4* from [8] is given in Table 1. This stream in algorithm-encoded form will be written as *3, 4, 6, <4, 3>, 2, <3, 6>, 4*;

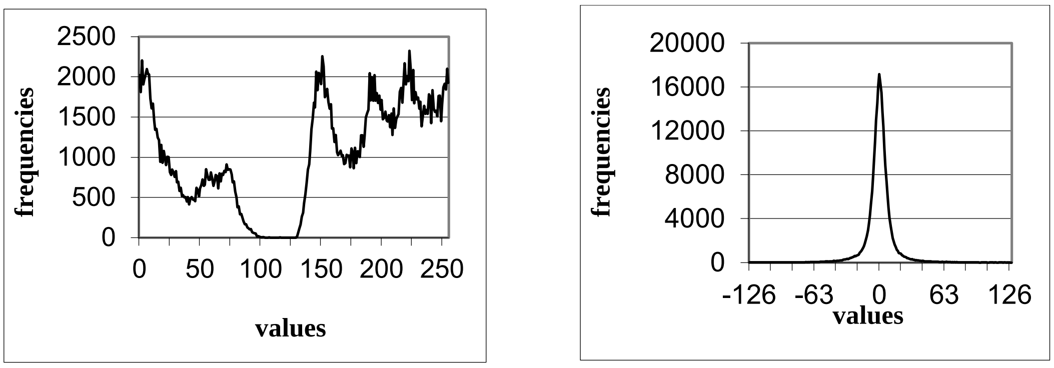
Table 1
Step-by-step flow compression results *3, 4, 6, 3, 4, 6, 3, 2, 6, 3, 4, 4* according to the algorithm LZ77 with [8]

№ step	Sliding window (input stream)		Matching sequence	Encoded data (output stream)	
	dictionary	buffer		<length, offset>	element
1.	-	3, 4, 6, 3, 4, 6, 3, 2, 6, 3, 4, 4	-	-	3
2.	3	4, 6, 3, 4, 6, 3, 2, 6, 3, 4, 4	-	-	4
3.	3, 4	6, 3, 4, 6, 3, 2, 6, 3, 4, 4	-	-	6
4.	3, 4, 6	3, 4, 6, 3, 2, 6, 3, 4, 4	3, 4, 6, 3	<4, 3>	-
5.	3, 4, 6, 3, 4, 6, 3	2, 6, 3, 4, 4	-	-	2
6.	3, 4, 6, 3, 4, 6, 3, 2	6, 3, 4, 4	6, 3, 4	<3, 6>	-
7.	3, 4, 6, 3, 4, 6, 3, 2, 6, 3, 4	4	-	-	4
8.	3, 4, 6, 3, 4, 6, 3, 2, 6, 3, 4, 4		-	-	-

2. At the second stage, the transition to an alternative color model is performed. For example, in a popular archiver WinRAR [9] before lossless image compression, the transition from the RGB color model to the R-G, G, B-G color model is performed;
3. On the third, the brightness of the pixel components is transformed using predictors [10], which during image traversal predict the brightness value of each component of the next pixel (for example, for the most common 24-bit images, these are the brightness of the red, green, and blue components, written as integers in separate bytes), using the brightness values of the same components of previously processed adjacent pixels. In the process of using this approach, **deviations** Δ_{ij} of the value of the brightness of the next pixel component $brightness_{ij}$ from the value predicted by the selected predictor $predict_{ij}$ are calculated and further coded, i.e.

$$\Delta_{ij} = brightness_{ij} - predict_{ij} \tag{1}$$

(i and j run respectively through all the rows and columns of the pixel components of the image). Adjacent image pixels most commonly have similar colors, which means that they also have close value brightness of relevant components, therefore value forecast often matches with value brightness of the regular components, most often it is close to this value and rarely – much differs from him (Figure 1 from [10]). That is, most of the values Δ_{ij} are close to zero. The use of predictors is similar to delta coding [2];



a) b)

Figure 1: A division of frequencies of values of green components of image Lena.bmp: a) before the use of predictor (H =7.59 of bpb); b) after the use of Left-predictor (H =5.34 of bpb) with [10]

4. At the fourth stage, context-independent coding forms element codes with lengths that dependent on their probabilities (processes code redundancy). Identical fragments or fragments of the same structure targeted by context-sensitive coding rarely occur in photo-realistic images, so the only universal step of lossless image compression is context-independent coding itself. It can even be used instead of context-sensitive luminance codes of individual pixels, if this further reduces the compression ratio (the ratio of the size of compressed to uncompressed image files, expressed in bpb, hereafter CR). According to our calculations, the application of a context-independent algorithm reduces the CR of images by 33% on average. Therefore, let's consider the ideas of this coding in more detail.

The main principle of context-independent coding is formulated as follows: **the length of the code of an arbitrary element with a higher probability should not exceed the length of the code of any element with a lower probability.** This principle is based on the fundamental position of information theory, according to which, in order to minimize the length of the sequence code, an element s_i with a probability of occurrence $p(s_i)$ should be coded with $-\log_2 p(s_i)$ bits [6]. Then the average code length of the block element after applying any context-independent algorithm according to the formula of Shannon [11] cannot be less than the *entropy of the source*

$$H = -\sum_i p(s_i) \times \log_2 p(s_i). \tag{1}$$

In practice today, two alternative context-independent methods are most often used: Huffman coding and arithmetic coding.

Huffman coding (HUFF) matches each element depending on its probability (frequency of occurrence) with a fixed prefix code [12] [13] [14][15]. This coding is most often performed in the following sequence: the probabilities (frequencies) of individual elements are calculated; elements in descending order of probabilities are arranged; the two elements with the lowest probabilities (most often the last ones in the created list) are iteratively combined in order to obtain one element, the code *0* to the first one, and *1* to the second one are added*;* the probabilities of the selected elements to calculate the probability of the formed element and insert this element into the sorted list of probabilities are summed; HUFF codes are formed, the formed codes in reverse order – from the top to each element are written.

The average length of the HUFF code coincides with the entropy of the source only when for all elements s_i the lengths of their optimal codes $-\log_2 p(s_i)$ are integers. In addition, the length of the HUFF code of even the most probable element cannot be less than one bit.

Arithmetic compression (ARIC), in contrast to HUFF coding, matches each successive element with an interval with a length proportional to its probability (frequency) [16][17][18][19]. At the same time, a subinterval corresponding to the first element of the flow is selected from the initial interval (most often [0; 1)) and further considered. This subinterval is again divided in proportion to the frequencies of individual elements, and the subinterval corresponding to the second element of the flow is selected from it (see, for example, Figure 2 from [20], on which the subintervals are ordered by the increasing values of the elements). The selection of subintervals continues in the same way until the end of the flow elements. Thus, the final length of the selected subinterval is equal to the product of the probabilities of all elements, and its beginning depends on the sequence of these elements in the stream. The result of ARIC is any number from the last subinterval received. Taking into account the finiteness of the numbers, to record the resulting number, the first significant digits of the intervals are monitored each time, and if they match, these digits are recorded in the output stream and excluded from further consideration. Decoding of the next ARIC element is performed by determining the corresponding next subinterval to which the read number belongs.

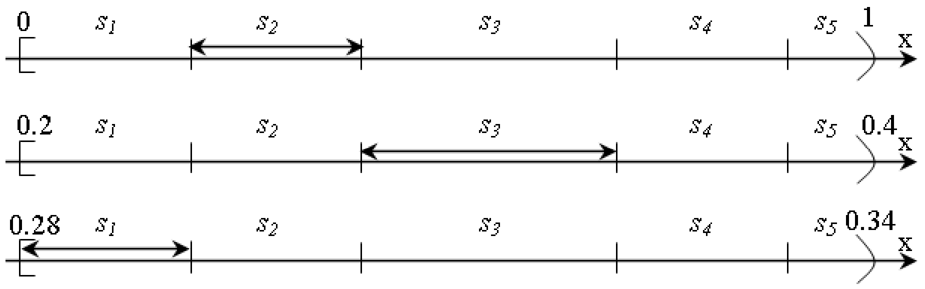


Figure 2 : Arithmetic coding of the first three elements of the sequence *36, 38, 35, 35, 40, 36, 40, 38, 45, 38* with [20]

From the description of the ARIC principles, it follows that in order to perform such arithmetic compression, the coder and, even more so, the decoder, must know the probabilities (frequencies) of individual elements. Today, two strategies for forming these frequencies are widely used: static and adaptive. In the case of a static strategy, the frequencies of individual elements are transmitted to the decoder in compressed data explicitly. When using an adaptive strategy, the intervals of the elements are formed synchronously by the encoder and decoder in the process of data processing. The adaptive strategy provides, as a rule, better CR (since it does not need to store the frequencies of individual elements in the compressed data and takes into account the unevenness of the distribution of the current sequence of elements, and not the flow in general), but it significantly slows down the work of the decompressor, because it requires recalculation of the probabilities of the elements in the decoding process. Graphics file formats should, first of all, provide fast decoding, so they often use a static strategy for forming element intervals.

To speed up the development of graphic formats, data compression formats are created, improved and used, which combine proven context-dependent and context-independent algorithms. One of the successful examples of such a combination is the DEFLATE dictionary compression format [21], which for processing of incoming flow uses context-dependent LZ77 algorithm [7], a the results of his work is compressed by Huffman's dynamic codes [12]. This format is used in many popular archivers (for example, GZIP [22]) and graphic formats (for example, PNG [23]) and does not need the acquisition licenses for use in applied software. We use this compressed data format to develop a lossless progressive hierarchical image compression format [8][10][20].

It is clear that implementations of the decoding algorithm of LZ77 codes must distinguish between individual literals and groups *<length; displacement>.* In the DEFLATE format, for this purpose, the lengths of the substitutions and the individual literals of the LZ77 algorithm are encoded together as numbers in the range [0; 285]. At the same time, in each compressed block numbers from the range [0; 255] correspond to the codes of individual literals, 256 marks the end of the block, and numbers from the range [257; 285] indicate the basic values of lengths. After the basic length values, there is a number of bits defined by the format, which, together with the basic value, uniquely determines the replacement length [21]. The offset is stored after the corresponding replacement length similarly – in the form of the base value and additional bits. The basic displacement value is within [0; 29]. In the DEFLATE format, the maximum encoded sequence length can be 258, the offset can be 32768, and different dynamic codes HUFF are used to encode the base literals/lengths and offsets.

The purpose of this article is to describe and justify the algorithm for redistribution of compressed data between modified DEFLATE blocks to improve the compression performance of this format.

3. Options for reducing the size of compressed data in the DEFLATE format

Let's first emphasize the reasons for improving compression by context-independent algorithms of the fourth stage in cases of applying the algorithms of the second and third stages. Let's investigate the dependence of entropy for the case of two elements (Figure 3 and [24]). Let the probability of the appearance of the first element be equal to p_1. Then the probability of the second element will be $p_2 = 1 - p_1$, and the entropy will be equal to $H = -p_1 \log_2(p_1) - (1 - p_1)\log_2(1 - p_1)$. We see that the entropy is maximal when $p_1 = p_2 = 0.5$. The greater the probabilities of the elements differ among themselves, that is, the greater the unevenness of the distribution is, the lower the entropy is [24]. For *M* the entropy of different elements is also maximal when the probabilities of these elements are the same and is calculated by Hartley's formula [24]:

$$H = \log_2 M. \tag{2}$$

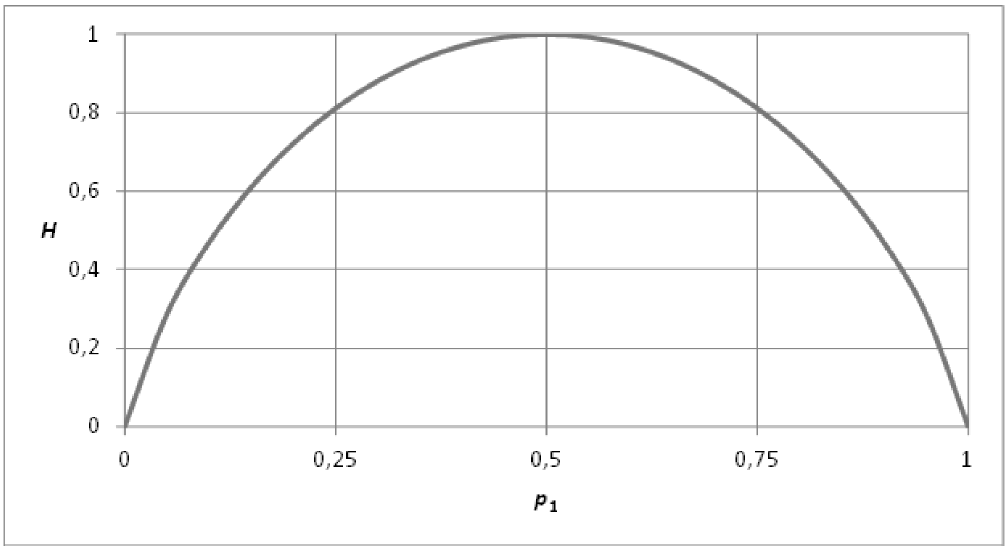


Figure 3: Dependence of the entropy of a source of two elements on the probability of the first element appearing

Difference color models of the second and predictors of the third stage of lossless image compression do not compress the image, but increase the unevenness of the brightness distribution around zero (Figure 1b) and therefore reduce the entropy (1), i.e. increase the code redundancy by reducing the inter-element one.

Taking into account the fact that arithmetic coding more accurately encodes elements whose occurrence of probabilities are not equal to powers of two [5], we modified the DEFLATE compressed data format by applying E. Shelvin's range coder in it, the code of which can be found, for example, in [25]. This encoder reads/writes bytes instead of bits of data and thus significantly speeds up encoding/decoding. For each element, it uses an interval with an integer length proportional to its probability. In order to store the length of the interval for each element in the header of the modified DEFLATE-block of compressed data, instead of the length of the HUFF code, we write the number of bits for storing the corresponding length of the interval in the general interval [0; 32768), and after the header – a binary record of this length without the first bit (which is always equal to one) [20]. The DEFLATE format uses a maximum of 285 different elements to encode literals/lengths replacement. If there are M different elements in the next modified DEFLATE block, then, taking into account (2), the number of additional bits for recording their interval lengths in the block header will be $M \times (\lceil \log_2 M \rceil - 1)$, if these elements are equally likely. In general, the header size of the modified DEFLATE block will not exceed 2635 bits.

The size of the most compressed data in each block by its *length entropy code* [20]

$$L_H = N \times H = N \log_2(N) - \sum_i N_i \log_2(N_i), \tag{3}$$

where N_i is the frequency of the element s_i in the next block, and $N = \sum_i N_i$.

At first glance, it seems that combining all the compressed data into one modified DEFLATE block should reduce CR, because then only one header of the compressed data block needs to be stored instead of many. But it was shown in work [20] that the combination of blocks of compressed data does not reduce (most often – increases) the length of their entropy code. The greater the difference between the relative frequencies of each element in the two input blocks is, the greater the additional increase in the length of the entropy code (Figure 4 from [20]) will be. This gain is zero only when the relative frequencies of the element in the two input blocks match.

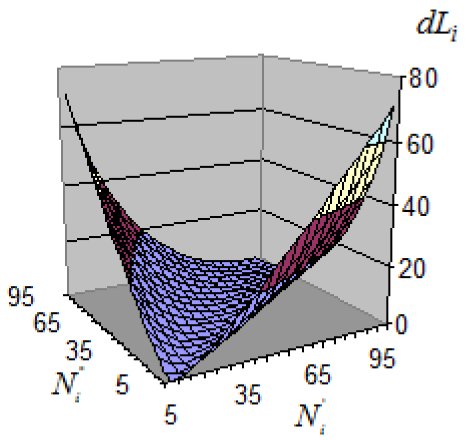


Figure 4: The dependence of the increase in the length of the entropy code of the combination of two input sequences on the frequencies of the element s_i in these input sequences [20]

Taking into account (3), the increase in the length of the entropy code due to the combination of two adjacent blocks will be calculated using the formula

$$dL_H = (N' + N'')\log_2(N' + N'') - N'\log_2(N') - N''\log_2(N'') - \\ - \sum_i (N_i' + N_i'')\log_2(N_i' + N_i'') + \sum_i N_i'\log_2(N_i') + \sum_i N_i''\log_2(N_i''), \tag{4}$$

where ' indicates the corresponding characteristic of the first block, and '' indicates the characteristic of the second.

Therefore, to reduce the size of compressed data in the DEFLATE format, it is advisable to implement the following options for redistributing compressed data between its blocks:

1. Combine two adjacent blocks of compressed data, if the increase in the length of the entropy code from their combination (4) does not exceed the savings from storing the header of the combined block instead of the two headers of these adjacent blocks;
2. Split blocks of compressed data into homogeneous blocks, if the decrease in the length of the entropy code from such a split exceeds the increase in the size of their headers.

4. Software implementation of redistribution of compressed data between modified DEFLATE-blocks

Let us first provide the text of the function in the C++ language for calculating the predicted length of the DEFLATE entropy code of the compressed data block (3) based on the frequencies of its elements, taking into account the predicted length of its header:

```
UBYTE4 countBitCodeFreqLL(UBYTE4 * masFreq, UBYTE4 freqElementMas)
{ double s=0; UBYTE2 countElement=0;
  for (UBYTE2 i=0; i< MaxLengthCodes; i++) // calculation of the subtractor from (3)
   if (masFreq [i]>0)
{ countElement++; // number of elements with non-zero frequency
     if (masFreq [i]>1) s+=masFreq [i]*log2(masFreq [i]); }
  return 70+countElement*log2(countElement)+ // predicted header length
  freqElementMas *log2(freqElementMas)-s; } // length of block entropy code
```

The function for calculating the predicted length of the combination of two blocks functions similarly:

```
UBYTE4 countBitCodeSumisnFreqLL (UBYTE4* masFreq1, UBYTE4* masFreq2,
UBYTE4 freqElementMas)
{ double s=0; UBYTE2 countElement=0;
  for (UBYTE2 i=0; i<MaxLengthCodes; i++)
   if (masFreq1[i]+masFreq2[i]>0)
{ countElement++;
     if (masFreq1[i]+masFreq2[i]>1)
s+=(masFreq1[i]+masFreq2[i])*log2(masFreq1[i]+masFreq2[i]); }
  return 70+countElement*log2(countElement)+freqElementMas*log2(freqElementMas)-s; }
```

Using these functions, a possible combination of two adjacent DEFLATE blocks (the first option of redistribution) is realized by postponing the data storage of the previous block of compressed data. If the sum of the sizes of the previous and current blocks does not exceed the maximum allowable and the combination of blocks reduces the size of the compressed data, then we will join the current block to the previous one. Otherwise, we will output the previous block, and postpone the current one for further analysis:

```
if (countPrevElement && countPrevElement+countCurrentElement < =OutputBufferSize)
// combination of previous and current DEFLATE blocks is possible
 { // calculate the frequency of literals / lengths of substitutions of the previous block
  memset (freqPrevBD, 0, MaxLengthCodes * sizeof (UBYTE4));
  for (j=0; j< countPrevElement; j++)
   if (codePrevLL [j]<256) freqPrevBD [codePrevLL [j]]++; // literal frequency
   else
    {LengthToCode (codePrevLL [j]-256, code, extra, value);
     freqPrevBD [code]++; } // frequency of the base value of the replacement length
  // calculation of size reduction from the combination of previous and current blocks
  countBitEconomUnion=countBitCodeFreqLL (freqCurrentBD, countCurrentElement)+
  countBitCodeFreqLL (freqPrevBD, countPrevElement) –
  countBitCodeSumisnFreqLL(freqCurrentBD, freqPrevBD, countCurrentElement+countPrevElement);}
// we combine the previous and current block, if it is appropriate and possible
if (countBitEconomUnion >=0 &&
    countPrevElement+countCurrentElement <=OutputBufferSize)
 {UBYTE2 *codeUnionLL=new UBYTE2[countPrevElement+countCurrentElement];
  UBYTE2 *codeUnionD=new UBYTE2[countPrevElement+countCurrentElement];
  memcpy(codeUnionLL, codePrevLL, countPrevElement * sizeof (UBYTE2));
  memcpy(codeUnionLL+countPrevElement,codeCurrentLL,countCurrentElement*sizeof(UBYTE2));
  memcpy(codeUnionD, codePrevD, countPrevElement * sizeof (UBYTE2));
  memcpy(codeUnionD+countPrevElement,codeCurrentD,countCurrentElement*sizeof(UBYTE2));
  UBYTE4 countUnionElement=countPrevElement+countCurrentElement;
  // after combining, we destroy the previous and current blocks
  if (countPrevElement)
   {delete [] codePrevLL; codePrevLL=NULL;
    delete [] codePrevD; codePrevD=NULL; }
  delete [] codeCurrentLL; codeCurrentLL=NULL;
  delete [] codeCurrentD; codeCurrentD=NULL;
  countCurrentElement=0;
  // postpone the combination to the previous block
  codePrevLL=codeUnionLL; codePrevD=codeUnionD;
  countPrevElement=countUnionElement;
  if (!lastblock) return; } // if not the last block, then we expect the next block
// otherwise, we output the previous block and postpone the current block
```

To perform the division of input blocks of compressed data into homogeneous blocks (the second variant of redistribution), we implemented the fragmentation algorithm described in [26], for dividing the input block into homogeneous blocks with different statistical characteristics. Having these homogeneous blocks, we iteratively choose and combine two adjacent ones that maximally reduce the length of the compressed data as long as such reduction is expected. Each of the remaining blocks after iterative combination is output as a separate DEFLATE block. Here is a fragment of the program for iteratively combining homogeneous blocks into initial DEFLATE blocks:

```
for (i=0; i<countRBlock; i++) // determine the predicted size of each homogeneous block
 {startPozRBlock=startRBlock [i];
  endPozRBlock=startRBlock [i+1]-1;
  // determine the frequency of literals/lengths of substitutions and offsets for the next block
  for (j=startPozRBlock; j< =endPozRBlock; j++)
   if (codeLL[j]<256) freqLLRBlock[i][codeLL[j]]++; // literal frequency
   else // frequency of replacement length and offset
    {LengthToCode(codeLL[j]-256, code, extra, value);
     freqLLRBlock[i][code]++; // accumulate the frequency of the literal/length of the substitution
     DistanceToCode(codeD[j], code, extra, value);
     freqDRBlock[i][code]++; } // accumulate the displacement frequency
  countBitCodeRBlock [i] =countBitCodeFreqLL (freqLLRBlock [i], endPozRBlock-
        startPozRBlock+2); } // predicted length of taking into account completion block
// define length combination of adjacent homogeneous blocks
for (i=0; i<countRBlock-1; i++)
 countBitCodeSRBlock[i]= countBitCodeSumisnFreqLL (freqLLRBlock [i], freqLLRBlock [i+1],
     startRBlock[i+2]-startRBlock[i]+1);// from taking into account code completion block
// combine adjacent homogeneous blocks for improvement compression
while (countRBlock >1)
 {economBit=countBitCodeRBlock[0]+countBitCodeRBlock[1]-countBitCodeSRBlock[0];
  indexMaxEconomic=0;
  for (i=1; i<countRBlock-1; i++)
   if ((int)(countBitCodeRBlock[i]+countBitCodeRBlock[i+1]-countBitCodeSRBlock[i])>economBit)
    {economBit=countBitCodeRBlock[i]+countBitCodeRBlock[i+1]-countBitCodeSRBlock[i];
     indexMaxEconomic=i; } // index of first of contiguous blocks improving compression
  if (ekonomBit <0) break; // combining blocks no longer reduces code length
  for (j=0; j< MaxLengthCodes; j++) // combine the frequencies of certain adjacent blocks
   freqLLRBlock[indexMaxEkonom][j]+=freqLLRBlock[indexMaxEkonom+1][j];
  // we shift the frequencies of the following blocks to the left
  memcpy(freqLLRBlock+indexMaxEkonom+1, freqLLRBlock+indexMaxEkonom+2,
    (countRBlock-indexMaxEkonom-2)* sizeof (freqLL));
  for (j=0; j<MaxDistanceCodes; j++) // process displacement block frequencies in the same way
   freqDRBlock[indexMaxEconomics][j]+=freqDRBlock[indexMaxEconomics+1][j];
  memcpy(freqDRBlock+indexMaxEkonom+1, freqDRBlock+indexMaxEkonom+2,
    (countRBlock-indexMaxEkonom-2)* sizeof (freqD));
  // shift the beginnings of the following blocks to the left
  memcpy(startRBlock+indexMaxEkonom+1, startRBlock+indexMaxEkonom+2,
    (countRBlock-indexMaxEkonom-1)* sizeof (UBYTE4));
  // determine the size of the combined block
  countBitCodeRBlock[indexMaxEkonom]=countBitCodeFreqLL(freqLLRBlock[indexMaxEkonom],
     startRBlock [indexMaxEkonom+1]-startRBlock[indexMaxEkonom]+1);
  memcpy (countBitCodeRBlock+indexMaxEkonom+1, countBitCodeRBlock+
    indexMaxEkonom+2, (countRBlock-indexMaxEkonom-2)* sizeof (UBYTE4));
  // it is necessary to recalculate the length of the combination with the previous block
  if (indexMaxEkonom>0)
   countBitCodeSRBlock[indexMaxEkonom-1]=countBitCodeSumisnFreqLL(
     freqLLRBlock [indexMaxEkonom-1], freqLLRBlock [indexMaxEkonom],
     startRBlock [indexMaxEkonom+1]-startRBlock[indexMaxEkonom-1]+1);
  if (indexMaxEkonom+2< countRBlock) // count the combination with the next block
   countBitCodeSRBlock[indexMaxEkonom]=countBitCodeSumisnFreqLL(
     freqLLRBlock [indexMaxEkonom], freqLLRBlock [indexMaxEkonom+1],
     startRBlock [indexMaxEkonom+2]-startRBlock[indexMaxEkonom]+1);
  // shift the lengths of the following combinations to the left
  if (indexMaxEkonom+3<countRBlock)
   memcpy(countBitCodeSRBlock+indexMaxEkonom+1, countBitCodeSRBlock+
     indexMaxEkonom+2, (countRBlock-indexMaxEkonom-3)* sizeof (UBYTE4));
  countRBlock--; }
  // then each received block is displayed as an output DEFLATE block
```

5. Results of the applying redistribution of compressed data between modified DEFLATE-blocks in the process of progressive hierarchical lossless image compression

We present the results of the impact of the redistribution of compressed data between DEFLATE-blocks (Table 2 – Table 4) on image compression of the well-known Archive Comparison Test test set (ACT, Figure 4) in the process of progressive hierarchical bypass [8][10][20]. You can download these images, for example, from [27]. This set contains both synthesized (№ 1, 2, 7) and photorealistic (the rest) images. The choice of this particular test set is determined by the diversity of its images and the availability in open sources of the results of testing algorithms on it by other researchers. Testing was conducted on a computer with an Intel processor Pentium 4 with a clock frequency of 3 GHz and 4 Gb of RAM.

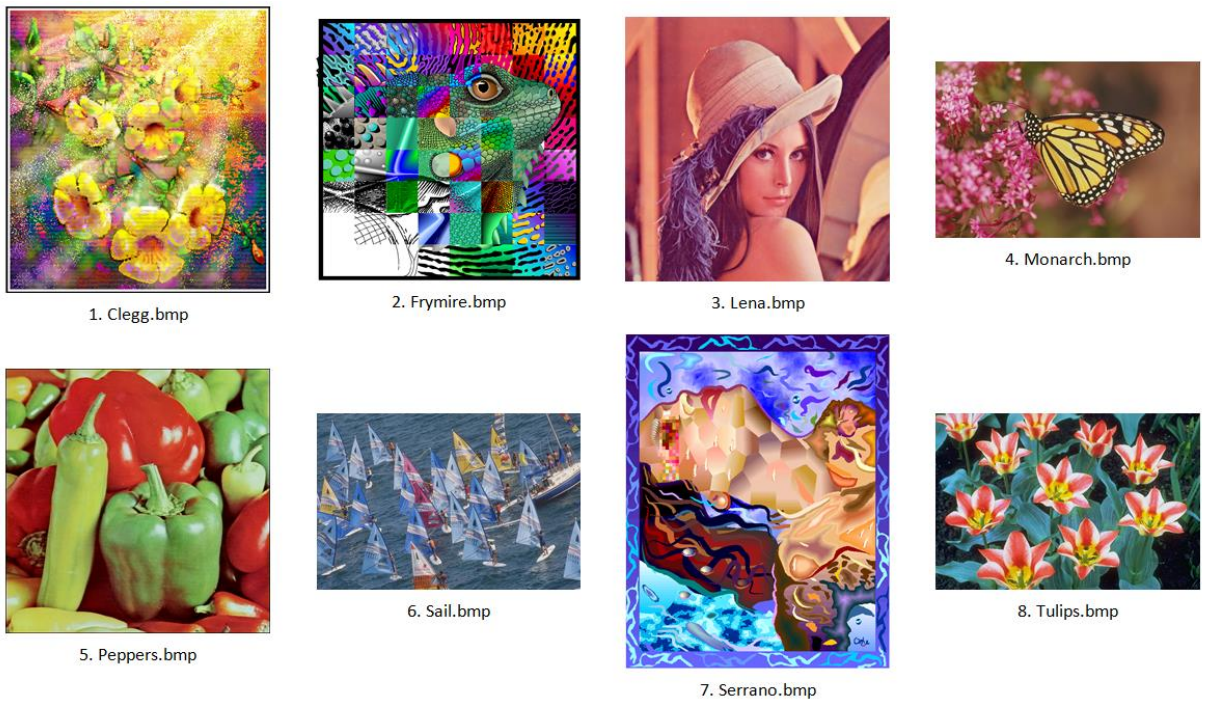


Figure 4: Image of ACT test set from [27]

Table 2
Sizes of compressed images of the ACT set after applying different variants of modified DEFLATE blocks, Kb

A variant of DEFLATE blocks	№ file								Overall size
	1	2	3	4	5	6	7	8	
No redistribution of compressed data	351	256	428	468	353	584	108	519	3067
With redistribution of compressed data	350	256	428	467	353	584	107	519	3064

Table 3
Encoding time of ACT set images using different variants of modified DEFLATE blocks, s

A variant of DEFLATE blocks	№ file								Middle time
	1	2	3	4	5	6	7	8	
No redistribution of compressed data	2.23	3.42	1.24	1.99	1.39	2.14	1.45	1.89	1.97
With redistribution of compressed data	2.30	3.44	1.44	2.19	1.48	2.33	1.49	2.10	2.10

Table 4
Decoding time of ACT coded images using various variants of modified DEFLATE blocks, s

A variant of DEFLATE blocks	№ file								Middle time
	1	2	3	4	5	6	7	8	
No redistribution of compressed data	0.70	1.30	0.31	0.44	0.42	0.45	0.52	0.53	0.58
With redistribution of compressed data	0.72	1.24	0.31	0.49	0.42	0.47	0.39	0.53	0.57

6. Discussions

From the data in Table 2-Table 4 we can see, that the application of redistribution of compressed data between modified DEFLATE blocks made it possible to reduce the total size of compressed images of the ACT set by 3 Kb, although on average it slowed down the encoding by 6.48%. Moreover, a reduction in the size of compressed images is observed both for photorealistic images (№ 4) and for synthesized images (№ 1, 7). Other compressed images are hundreds of bytes smaller. Such modest results are explained by the fact that, firstly, the maximum gain from the combination of adjacent modified DEFLATE blocks is equal to the block header (up to 380 bytes), and secondly, after using difference color models and predictors, these images have a high uneven distribution of element frequencies (Figure 1b) and partitioning does not significantly increase their homogeneity and, thirdly, the data of different layers and passes of progressive hierarchical compression are stored in different compressed blocks by default. On the other hand, the redistribution of compressed data between DEFLATE blocks made it possible to speed up decoding by an average of 2.14% due to discrete-tone images, which indicates the feasibility of its use. For example, we use the redistribution of compressed data between modified DEFLATE-blocks in our developed HBF-LS lossless progressive hierarchical image compression format [8][10][20] when the encoding time is insignificant.

7. Conclusions

1. Using compressed data formats, which provide for their division into blocks, it is worth analyzing the expediency of combining adjacent blocks and dividing blocks into homogeneous fragments to reduce CR before writing these blocks into a file;
2. The combination of DEFLATE blocks reduces the size of compressed data by a maximum of the size of the header of one such block, and splitting increases the uneven distribution of the relative frequencies of their elements, so these actions should be implemented in image compression programs;
3. The combination of DEFLATE blocks of compressed data makes it possible to speed up decoding by reducing the number of generation intervals or codes of individual elements for each such block.

In the future, we plan to improve the mechanism of the "lazy" layout of the LZ77 algorithm [8] by memorizing and using smaller offsets to shorter identical sequences that can be applied in the process of its formation, and to investigate the effectiveness of using the previous layout this algorithm to predict the lengths of literals, lengths of substitutions and offsets.

References

[1] G. Wallace, ”The JPEG still picture compression standard,” Communication of ACM, 34 (1991) 30-44. doi:10.1145/103085.103089.

[2] O. Shehata, “Unraveling the JPEG,” Parametric Press, 1 (2019). URL: https://parametric.press/issue-01/unraveling-the-jpeg.

[3] T. Guo, T. Zhang, E. Lim, M. López-Benítez, F. Ma, L. Yu, “A review of wavelet analysis and its applications: Challenges and opportunities,” IEEE Access, 10 (2022), 58869-58903. doi:10.1109/ACCESS.2022.3179517.

[4] H. D. Kotha, M. Tummanapally and V. K. Upadhyay, “Review on Lossless Compression Techniques,” Journal of Physics 1228 (2019). doi:10.1088/1742-6596/1228/1/012007.

[5] D. Selomon, A Guide to Data Compression Methods, Springer, New York, 2002, 295 p. doi:10.1007/978-0-387-21708-6.

[6] R. Gonzalez, R. Woods, Digital Image Processing, Fourth Edition., Pearson, London, 2017, 1192 p. ISBN 9353062985.

[7] J. Ziv, A. Lempel, “A universal algorithm for sequential data compression,” IEEE Transactions on Information Theory, 23(3) (1977) 337-343.

[8] A. Shportko, A. Bomba, V. Postolatii, Rejection of the Inefficient Replacements while Forming the Schedule of the Modified Algorithm LZ77 in the Process of Progressive Hierarchical Compression of Images without Losses, in: Proceedings of the 6th International Conference Computational Linguistics and Intelligent Systems (COLINS 2022), Glivice, Poland, 12-13 May 2022, volume 3171, pp. 1594-1605. URL: http://ceur-ws.org/Vol-3171/paper113.pdf.

[9] WinRAR download free and support, version 7.00, 2024. URL: https://www.win-rar.com/start.html?&L=0.

[10] A. Shportko, V. Postolatii, Development of Predictors to Increase the Efficiency of Progressive Hierarchic Context-Independent Compression of Images Without Losses, in: Proceedings of the 5th International Conference Computational Linguistics and Intelligent Systems (COLINS 2021), Kharkiv, Ukraine, Apr. 22-23, 2021, volume 1, pp. 1026-1038. URL: http://ceur-ws.org/Vol-2870/paper77.pdf.

[11] C. E. Shannon, “A Mathematical Theory of Communication,” Bell System Technical Journal, 27 (1948) 379-423, 623-656. doi:10.1002/j.1538-7305.1948.tb00917.x.

[12] D. Huffman, “A Method for the Construction of Minimum Redundancy Codes,” Proceedings of the IRE, 40(9) (1952) 1098-1101. doi:10.1109/JRPROC.1952.273898.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Fourth Edition, MIT Press, 2022, pp. 431–439. URL: http://mitpress.mit.edu/9780262046305/introduction-to-algorithms.

[14] J. Duda, K. Tahboub, N. J. Gadil and E. J. Delp, “The use of asymmetric numeral systems as an accurate replacement for Huffman coding,” Picture Coding Symposium, Cairns, QLD, Australia, Jul. 30, 2015. doi:10.1109/PCS.2015.7170048.

[15] M. A. Rahman, M. Hamada, “Lossless image compression techniques: A state-of-the-art survey,” Symmetry, 11 (2019) 1274. doi:10.3390/sym11101274.

[16] J. Rissanen, “Generalized Kraft Inequality and Arithmetic Coding,” IBM Journal of Research and Development, 20(3) (1976) 198–203. doi:10.1147/rd.203.0198.

[17] J. Rissanen, G. G. Langdon, “Arithmetic coding,” IBM Journal of Research and Development, 23(2) (1979) 149–162. doi:10.1147/rd.232.0149.

[18] I. H. Witten, R. M. Neal and J. G. Cleary, “Arithmetic Coding for Data Compression,” Communications of the ACM, 30(6) (1987) 520–540. doi:10.1145/214762.214771.

[19] A. Moffat, R. M. Neal and I. H. Witten, “Arithmetic coding revisited,” ACM Transactions on Information Systems, 16(3) (1998) 256-294. doi:10.1145/290159.290162.

[20] A. Ya. Bomba, A. V. Shportko and A. V. Shportko, “Features of the application of arithmetic coding in the process of lossless progressive hierarchical compression of images,” Proceedings of the National University "Lviv Polytechnic", Series: Information Systems and Networks, 783 (2014) 12-22. URL: http://nbuv.gov.ua/UJRN/VNULPICM_2014_783_4.

[21] P. Deutsch, DEFLATE Compressed Data Format Specification, version 1.3, RFC 1951. URL: https://www.rfc-editor.org/rfc/rfc1951.

[22] P. Deutsch, J-L. Gailly, ZLIB Compressed Data Format Specification, version 3.3, RFC 1950, Network Working Group, 1996, 10 p. URL: http://www.ietf.org/rfc/rfc1950.txt.

[23] J. Miano, Compressed Image File Format: JPEG, PNG, GIF, XBM, BMP, Addison Wesley, New York, 1999, 264 p. ISBN 0201604434.

[24] Yu. P. Zhurakovskyi, V. P. Poltorak, Teoriia informatsii ta koduvannia, Vyshcha shkola, Kyiv, 2001, pp. 33. URL: https://b.eruditor.link/file/348269.

[25] Compression by PPM + Arithmetic Coder C# implementation (based on C++ algorithm from the book "Data Compression Methods"), 2022. URL: https://gist.github.com/Geograph-us/a1beae713c337243c478cb5575a22f75.

[26] Shportko A. V. Rise of efficiency of compression of coloured images in the PNG format, Ph.D. thesis, Rivne State University of the Humanities, Rivne, 2010, pp. 83-85. URL: https://dspace.megu.edu.ua:8443/jspui/handle/123456789/1665.

[27] ACT – Test Files, 2002. URL: http://www.compression.ca/act/act-files.html.