

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

Alexander Shportko¹, Andriy Bomba² and Veronika Postolatii³

¹ *Academician Stepan Demianchuk International University of Economics and Humanities, 4, Acad. S. Demianchuk Str, 33000, Rivne, Ukraine*

² *National University of Water and Environmental Engineering, 11, Soborna Str, 33028, Rivne, Ukraine*

³ *National University "Lviv Polytechnic", 12, St. Bandera Str, 79000, Lviv, Ukraine*

Abstract

The modification of the algorithm of the lexical compression LZ77 for increasing the efficiency of the progressive hierarchical compression with the use of additional search of identical sequences of the nearest pixels that have been processed before is offered. Variants of rejection of inefficient substitutions while forming such a schedule are listed. The results of using the suggested algorithm and variants of rejection of the substitutions for compression of the images of ACT set are demonstrated. It is shown, for example, the elimination of overlays of substitutions among the nearest adjacent pixels allows to improve the compression.

Keywords

Progressive image compression, lossless compression, dictionary compression methods, LZ77 algorithm schedules.

1. Introduction

Today, due to the rapid development of communication technologies and increasing information needs of society, the problem of data compression is still relevant. After all, data compression allows to increase the speed of information exchange over the network and decrease the use of disk space proportionally [1]. It is well known that data compression is possible by reducing the redundancies. The more types of redundancies the algorithm detects and reduces, the more efficient the compression is. To reduce the level of cross-element redundancy in image file formats and archiver vocabulary data compression algorithms because they provide the fastest decoding [2, c. 82]. Therefore, increasing the efficiency of dictionary algorithms in general and the most popular among them is LZ77 algorithm [3] in particular is an urgent task nowadays.

2. Related works

The Deflate dictionary compression format was developed by Phil Katz in 1993 for the second version of the PKZIP archive and formalized in RFC 1951 [4] in 1996. Compression in this format is performed by the context-sensitive algorithm LZH [2, p. 90-91], which is a variant of the LZ77 algorithm, and the context-independent Huffman algorithm [5]. Today, this format is successfully used not only in many ZIP archives, but also in the popular graphic format PNG, which is actually the standard for saving images without losses. In addition, the Deflate format does not require the purchase of licenses for the use in software application [2, p. 95]. Therefore in this article we will

COLINS-2022: 6th International Conference on Computational Linguistics and Intelligent Systems, May 12–13, 2022, Gliwice, Poland

EMAIL: ITShportko@ukr.net (A. Shportko); ABomba@ukr.net (A. Bomba); VeronikaShportko@gmail.com (V. Postolatii)

ORCID: 0000-0002-4013-3057 (A. Shportko); 0000-0001-5528-4192 (A. Bomba); 0000-0002-9460-0781 (V. Postolatii)



© 2022 Copyright for this paper by its authors.

Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

propose modifications of the LZ77 algorithm and Deflate format for progressive hierarchical lossless image compression.

3. Application of the LZ77 algorithm for sequential lossless image compression

The dictionary algorithm eliminates redundancy between the same data fragments, keeping references to the same data when duplicates are detected. Describing dictionary algorithms, a fixed number of previously encoded indivisible elements (literals) of the input stream is called a *dictionary*, and subsequent uncoded – a *buffer*. Totality dictionary with encoded literals and buffer with uncoded are also called *sliding windows*, because they are constantly moving synchronously on the elements of the stream.

Algorithm LZ77 [3] (in the context of the dictionary compression format Deflate [4]) is based on the replacement in the encoding process in the output stream of the sequence of consecutive literals of the buffer with reference to a similar sequence of vocabulary literals in the form of a pair of numbers $\langle \text{length}, \text{offset from the end of the dictionary} \rangle$. In the absence of a similar sequence of literals in the dictionary longer than two literals, the first literal 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 encoding continues similarly until the end of the literals of the input stream. The same sequence may extend beyond the dictionary to the buffer area, but must begin in the dictionary. It is clear that the more and longer replacements $\langle \text{length}, \text{offset} \rangle$ can be found in the encoding process – the better the compression ratio will be.

When decoding the codes of the LZ77 algorithm, individual literals are copied to the output stream without changes. Pairs $w \langle \text{length}, \text{offsets} \rangle$ are decoded by sequentially copying from the end of the output stream at the specified offset to the end of the output stream the required number of literals.

Naturally, the LZ77 code decoding algorithm must distinguish between individual literals and pairs $\langle \text{length}, \text{offset} \rangle$. In the Deflate format for this purpose, the substitution lengths and individual literals of the LZ77 algorithm are coded together by numbers within [0; 285] (as in the LZH algorithm [2, pp. 90-91]). The numbers from the range [0; 255] correspond to the codes of individual literals, 256 indicates the end of the block, and numbers in the range [257; 285] indicate the basic values of lengths. After the base values of the lengths there is an additional number of bits (up to five) determined by the format, which together with the base value unambiguously determines the length of the replacement [2, p. 98]. The offset is saved immediately after the corresponding replacement length in the same way - in the form of a base value and additional bits (up to 13). The base value of the offset is within [0; 29] [2, p. 99]. To eliminate code redundancy, the elements of the distributions of literals/base values of lengths and base values of offsets are encoded by Huffman codes (hereinafter HUFF) [5]. For example, the same sequence of 20 components at the beginning of the buffer and also contained in the dictionary of offset 24 components in Deflate format is encoded by the HUFF code of the base value 269 of the literal / substitution length distribution and two additional substitution length bits and the HUFF code of the base value 8 offset distribution and three additional offset bits. It is clear that longer substitutions occur less often than shorter ones, just as larger offsets occur less often than smaller ones, so the technology of base values and additional bits actually groups elements with lower probabilities and thus speeds up their HUFF code. In the Deflate format, the maximum value of the length of the encoded sequence can reach 258, the offset – 32768, and to encode literals/base values of lengths and base values of offsets, different HUFF codes are used.

Consider the example of using the LZ77 algorithm for sequential lossless image compression, which is most commonly used today. Most images are now stored in the RGB color model with an 8-bit sampling accuracy. That is, the color of each pixel is set by three bytes, which consistently contain the brightness of its red, green and blue components. The sequence of sequentially bypassing the brightness of the pixel components in this color model is schematically shown in Figure 1.

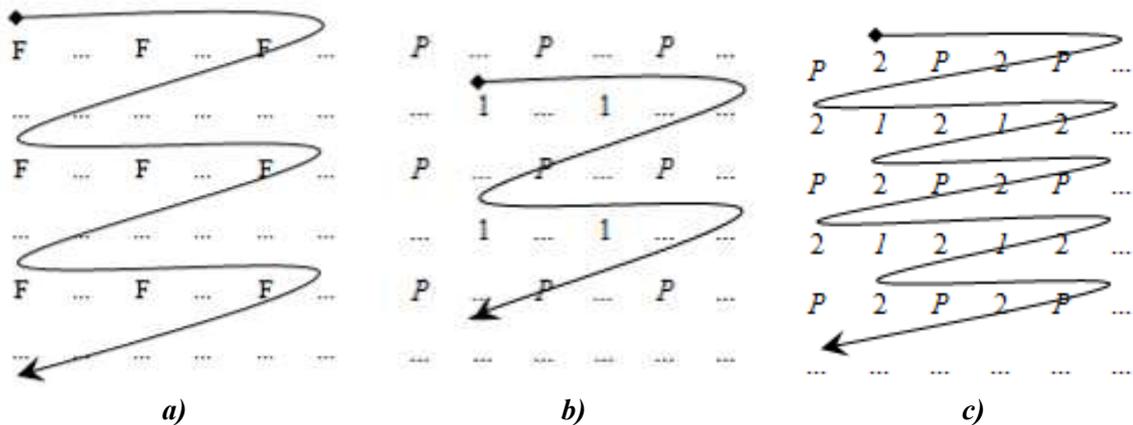


Figure 2: The order of pixel traversal in the process of progressive hierarchical processing: a) pixels of the first layer, b) pixels of the first pass of the next layer, c) pixels of the second pass of the next layer

In the following layers ($l = \overline{2, k+1}$) intermediate pixels of the image are processed in two passes: the first is sequentially processed those that are contained at the intersection of diagonals of squares with vertices in adjacent pixels of the previous layers with a step $h_l = 2^{k+2-l}$ both in rows and in columns (see Figure 2b), and on the second row pixels successively pass between adjacent pixels of previous layers and pixels of the first pass with the same step on columns and with twice reduced - on lines (see Figure 2c). On Figure 2 the symbol F indicates the pixels of the first layer, symbol P – pixels of the previous layers, number 1 – pixels of the first pass of the next layer, number 2 – pixels of the second pass of the next layer. Pixels that have been processed before and therefore are not processed in the next pass of the layer are highlighted in italics.

Sequentially placed data of all passes form an input stream for encoding by the LZ77 algorithm. The data of the next layer increases the number of processed pixels by about 4 times. Therefore, it is possible to stop the decoding process using this bypass scheme after filling the output area, without processing the codes to the end. However, the proposed sequence of the image pixel traversal allows not only to speed up decoding when the size of the output area is much smaller than the image size, but also to use hierarchical predictors to predict the value of each element of the next pixel [7].

On the other hand, bypassing pixels of the image in several passes increases the compression ratios of the LZ77 algorithm relative to sequential traversal, because in the process of progressive traversal for unencoded brightness of each pixel of the buffer corresponding brightness of adjacent pixels, which are encoded by a significant number of bits. In particular, on the first layer pixels are processed with a step $h_1 = 2^k$ (see Figure 2a), therefore, the brightness of adjacent pixels will not be included in the dictionary at all. The existence of the large identical fragments in the image, removed at this step, is questionable, so the first layer of the LZ77 algorithm is inefficient. With each subsequent layer after the second pixel traversal step is halved and therefore the probability of the same sequences in the buffer and dictionary increases, but for the next pixel buffer brightness adjacent pixels images will either be processed on subsequent layers and therefore not included in the dictionary or already processed on the previous layers or passages, and therefore have large offsets in the dictionary.

5. Modification of the LZ77 algorithm for progressive hierarchical compression of images without losses

To increase the efficiency of the LZ77 algorithm in the process of progressive hierarchical traversal on all layers, starting from the second, we will search for the same sequences for buffer elements not only in the dictionary, **but also starting from the nearest previously processed pixels with the next layer**. Because among the previously processed pixels is another pixel X has the highest level of correlation with the nearest among them, the smallest 6 offsets (with codes from 1 to

6) will be fixed on the nearest previously processed pixels (Figure 3), and to ensure the uniqueness of the decoding offset codes in the dictionary increase by 6.



Figure 3: Offset codes to the brightness of adjacent previously processed pixels on the layers, starting with the second: a) for the first pass; b) for the second pass

In the first passes, the four closest contiguous previously processed pixels are removed diagonally (see Figure 3a). These are pixels from the previous layers (see Figure 2b) and therefore they have large offsets in the dictionary, and the same sequences in the image that start with them can generally be scattered throughout the dictionary. The fifth and sixth offsets for the first pass encode the nearest processed pixels of the same pass horizontally and vertically, respectively, but in the dictionary the offset of the nearest vertical pixels is increased by the number of pixels in the pass line.

In the second passages, the four nearest contiguous pixels are spaced horizontally and vertically (see Figure 3b). These are pixels from previous layers or passes (see Figure 2c). They are located closer to the next pixel than in the first pass, and therefore they start more identical sequences. Searching for the same sequences, starting with the nearest previously processed pixels of the previous layers, allows you to find matching sequences that are generally scattered on different layers in the dictionary and therefore are encoded on average by a larger number of bits.

The fifth and sixth offsets for the second pass are diagonally spaced and encode the pixels of the same pass from the previous line, but they have larger offsets in the dictionary. Symmetrical pixels relative to pixel *X* for offsets 5 and 6 are not encoded because they have not yet been processed in the next pass.

Most importantly, finding the same sequences from the nearest previously processed pixels reduces the compression ratio (hereinafter CR) on the last layer, because these pixels are contained in the image next to the next pixels of the *X* buffer and have the highest level of correlation with them. For example, during the first pass of the last layer for the brightness of the pixel components of the image from Figure 1, the same sequence among the nearest pixels (Figure 4) will be found at offset 3 (according to the codes of Figure 3a). In the dictionary, this same sequence is generally scattered in different places.

3, 4, 6	3, 4, 6	3, 2, 6	3, 4, 4	2, 1, 5	...
3, 4, 6	3, 2, 6	3, 4, 4	2, 1, 5	0, 2, 2	...
2, 1, 3	3, 2, 1	0, 2, 2	4, 2, 3	1, 1, 1	...
4, 1, 1	3, 4, 4	4, 3, 2	0, 0, 3	1, 1, 1	...
3, 4, 6	3, 2, 6	3, 4, 4	2, 1, 5	0, 2, 2	...
...

Figure 4: The same sequence of 6 components at offset 3 (shaded vertically) for the brightness of the pixels of the first pass of the last layer (shaded horizontally) conditional RGB-image

The pixels in this pass are processed with step 2 both in the image and in the same sequence. The total number of matching components in Fig. 4 is 6. Adjacent pixels to the left of the shaded ones are also the same, but in the process of progressive hierarchical compression they bypass the previous layers,

and therefore are not included in this replacement, although during a sequential traversal they would be included in one 12-component replacement (on Fig. 1 it is indicated by dotted arrows).

Therefore, additional search of identical sequences, starting from the nearest previously processed pixels, either finds such sequences that are generally scattered in the dictionary and thus increases the probability of finding these sequences, or uses smaller offsets than when searching only in the dictionary, and therefore reduces CR images.

6. Rejection of the inefficient substitutions in the process of forming schedule of the modified algorithm LZ77

As it was mentioned above, the modified schedule of the LZ77 algorithm should be formed in two passes: the first to look for replacements among the nearest previously processed pixels, and the second to find replacements in the dictionary for those pixel's components that are not replaced by the first pass. We will consider the algorithm for rejection of inefficient generated substitutions of the modified LZ77 schedule in [8]. In the same section, we will substantiate the options for refusing or extending replacements in the **process of forming** a modified LZ77 schedule, which predictably increase the length of the code.

6.1. Analysis of increments of additional bits of the offset codes

Substitutions among the nearest previously processed pixels (offset from 1 to 6) in addition to the context-independent code of the base value are encoded with a maximum of one additional bit [4]. But dictionary substitutions have larger offsets and their code can contain up to thirteen additional bits [2, c. 99]. Longer identical sequences are searched in the dictionary from right to left, so longer identical sequences for buffer literals have larger offsets and, as a result, are encoded with fewer additional bits.

On the other hand, the entropy of the pixel components after the application of combinations of hierarchical predictors for our discrete-tone images including ACT [9] is on average 2.02 bpb, for photorealistic – 4.54 bpb [7], but as it is impossible to determine the image type, we predict the value of entropy on average at 4 bpb. Therefore, we will assume that lossless image compression by a context-independent algorithm after using these combinations of predictors provides data compression by an average of 50 %. Therefore, in the process of forming a modified schedule of the LZ77 algorithm, it is likely that a longer identical sequence in the dictionary of 1, 2 or 3 literals, which without the LZ77 algorithm would be encoded on average in 4, 8, or 12 bits, respectively, will increase additional bits offset codes by 13. Thus, such an extension of the replacement will be impractical, as it will increase the length of the code. It is not a good idea to rely only on the algorithm of rejection of inefficient generated substitutions of the modified schedule LZ77 [8] the previous replacement could be effective (encoded with fewer bits than the literals it replaces), and the current extended - inefficient. The algorithm [8] will reject this extended replacement and return all replaced literals, but will not return the previous effective replacement. Therefore, inefficient replacements should be discarded both during and after the formation of each Deflate block. The fragment of the subroutine in the C++ language to reject ineffective substitutions based on the analysis of increments of additional bits of the offset codes may be as follows:

```
if (requiredReplace // if the replacement is still considered to be effective
    && bestlength>0 // and there is a previous replacement (the current extends the previous one)
    && bestPos-current->index+delta>180 // and the distance between the previous and
    // current replacements from 60 pixels (4 additional offset bits possible)
    && countlength-bestlength<=3 // extensions no more than 3 literals
    && currentPosImage-bestPos<=3054) // 3054/3+plusOffset=1024 –
    // the previous offset had up to 8 additional bits
{deltaReplace=ceil(log2((currentPosImage-current->index+delta)/3.0+
    plusOffset))-2; // number of additional bits of the current extended replacement
// to calculate the increments of additional bits of the offset codes
```

```

// subtract the number of additional bits of the previous replacement
if (bestPos!=currentPosImage) // the previous replacement was also in the dictionary
    deltaReplace=ceil(log2((currentPosImage-bestPos)/3.0+plusOffset))-2;
else // the previous replacement was based on the nearest previously processed pixels
    if (bestoffset>4) deltaReplace--; // additional offset bits 5 and 6
lenPlusLiteral=4*(countlength-bestlength); // predicted code length extension replacement
// (increase in length) without using the LZ77 algorithm
if (lenPlusLiteral<deltaReplace) // if the code length of the literals is less
// increment of additional bits of the offset code
    requiredReplace = false; } // the current replacement extension is ineffective

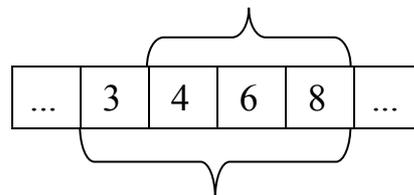
```

6.2. Elimination of overlays of dictionary replacements with replacements among the nearest previously processed pixels

In the process of forming a modified schedule of the LZ77 algorithm, it may happen that the replacement of the dictionary on the second pass will overlap with the replacement among the nearest previously processed pixels of the first pass. This will not only reduce the replacement of the first pass, but may eliminate it altogether if it is shorter than three literals. Dictionary substitutions have more substitutions and not fewer extra bits of code than the nearest processed pixel substitutions, so this overlay can increase code length and be ineffective. Therefore, each dictionary overlay on the next pixel replacement should be eliminated by reducing the length of the dictionary override. Therefore, each overlay of the dictionary replacement on the part of the replacement of the next pixels should be eliminated by reducing the length of the dictionary replacement. In addition, for a read replacement for previously processed pixels, it is not even advisable to look for an extended replacement for the dictionary, if it is followed by a replacement for previously processed pixels.

In addition, there are situations when the dictionary replacement completely absorbs the effective replacement of the nearest previously processed pixels and several adjacent literals (Figure 5), but is rejected by the analysis of the generated Deflate-block algorithm in [8].

Replacement on the next processed pixels



Replacement according to the dictionary

Figure 5: Example of absorption by substitution according to the dictionary of effective substitution by the nearest previously processed pixels

Then the compressed data encodes literals that correspond to a dictionary replacement, rather than an effective replacement for the nearest previously processed pixels and related literals. Thus, it increases the length of the code. Therefore, the dictionary replacement, which may be ineffective after the formation of the Deflate-block (in our implementation of the compressor - shorter than 9 literals) and contains replacements for the nearest previously processed pixels, it is also advisable to reduce the separation of first pass replacements. We separated the substitutions by the nearest processed pixels, if they were placed at the beginning or end (see Figure 5) dictionary replacements. If the replacement of the first pass was located in the middle of the replacement of the second pass, we did not reduce the replacement in the dictionary, because it covers several unencoded literals in the first pass to the left and right of the replacement of the next processed pixels.

To form the schedule of the modified LZ77 algorithm, we use an additional byte array *offsetAdjacentPixel*, in which for each literal (brightness of the components of each pixel) save the offset to the next identical previously processed pixels (1 to 6) or 0 if no pixels. This array is formed

during the first pass of this algorithm on the principle of «greedy» decomposition, because compression on the nearest processed pixels is often more effective than compression on the dictionary. Using the array *offsetAdjacentPixel* allows in the second pass of the modified algorithm LZ77 to eliminate the overlap of replacements in the dictionary for replacements on the nearest previously processed pixels, as described above. The fragment of the program in C++ to eliminate such overlays can be as the following:

```
// check for two adjacent replacements on the next pixels
if (offsetAdjacentPixel[currentPozImage]!=0) // there is a replacement for the next pixels
    {readReplacImage(lenAdjacent, offsetAdjacent);
    bestlength=lenAdjacent;
    bestoffset=offsetAdjacent;
    // if after the current replacement there is a replacement for the next processed pixels
    if (currentPosImage+bestlength < countByteImage &&
        offsetAdjacentPixel[currentPosImage+bestlength]!=0)
        return; } // then we are not looking for an extended replacement in the dictionary

// eliminate the overlap on the replacement part by adjacent pixels
if (currentPosImage+countlength<countByteImage &&
    offsetAdjacentPixel[currentPosImage+countlength-1]!=0)
    {// if the last literal replacement is in the dictionary to be replaced by the next pixels
    while (countlength>bestlength && // while the current replacement is longer than the
        // previous one and the last literal is to be replaced by the next pixels
        offsetAdjacentPixel[currentPosImage+countlength-1]==
        offsetAdjacentPixel[currentPosImage+countlength])
        countlength--; // reduce the replacement in the dictionary
    if (countlength==bestlength) // if reduced to the previous replacement
        requiredReplace=false; } // then reject the current replacement

// look for and separate the replacement by the next pixels
// if dictionary replacement may be ineffective (shorter than 9 literals)
if (requiredReplace && countlength<minLenReplaceLiteral)
    {// if at the beginning of the dictionary replacement there is no replacement for the next pixels
    if (offsetAdjacentPixel[currentPosImage]==0)
        {// reject the right replacement by adjacent pixels
        while (countlength>bestlength &&
            offsetAdjacentPixel[currentPosImage+countlength-1]!=0) countlength--;
        if (countlength==bestlength)
            requiredReplace=false; } // replacement expansion did not occur
    else // if at the beginning of the dictionary replacement a replacement for the next ones
        requiredReplace=false; } // is detected pixels, then leave the detected replacement
```

6.3. Elimination of the dictionary substitutions using predicted entropy code lengths

As it was noted above, the replacement of the $\langle \text{length}, \text{offset} \rangle$ of the modified LZ77 algorithm should be considered to be effective if it is encoded with fewer bits than the length of the literal codes it replaces. In this subsection we will consider an algorithm for rejection of the dictionary substitutions in the process of scheduling a modified LZ77 algorithm if they are ineffective to predicted entropy code lengths.

Context-independent coding applied to the distribution of literals / base values of substitution lengths and the distribution of base values of offsets is based on the fundamental position of information theory, according to which to minimize the length of the sequence code each value of the distribution element i with the probability of occurrence p_i it is advisable to encode $l_i = -\log_2 p_i$ bits [2, p. 17], and therefore average code length of the block element after the application of any context-independent algorithm, according to the formula of Shannon [10, c. 621], cannot be less than

the *entropy of the source* $H = -\sum_i p_i \times \log_2 p_i$. So let's l_i call the length of the entropy code of the element i . For context-independent coding of literals, substitution lengths and offsets of the modified LZ77 algorithm, we modified the Deflate format, using arithmetic coding (hereinafter ARIC) [2, p. 35-43; 11] instead of HUFF coding, because the average length of the ARIC code is close to entropy [12, p. 76]. This made it possible to use the length of the entropy code to estimate the code length of each distribution element.

Let each of the values i (brightness of an individual component or the base value of context-sensitive code), occurs n_i times in sequence length $N = \sum_i n_i$. According to the statistical definition of probability, $p_i = n_i / N$, therefore, the length of the entropy code of the element is

$$l_i = -\log_2 p_i = \log_2 \frac{N}{n_i}. \quad (1)$$

But the entropy (average code length) of literals decreases with increasing layer number [7] and substitution effective on the previous layer or pass may be ineffective for the current layer or pass. Therefore, after the first pass of the modified algorithm LZ77 on the nearest processed pixels and the formation of the array *offsetAdjacentPixel* for each layer and pixel pass of the image, we predict the lengths of the literal/length distribution codes to determine the effectiveness of dictionary substitutions during the second pass of this algorithm. The lengths of the distribution codes of the base values of the offset offsets in the dictionary are not predicted, because they can not be determined in advance.

A subroutine for predicting the lengths of distribution codes literal/substitution lengths *lenPrognozLLBlockFilter* by formula (1) in language C++ for each pass of hierarchical processing can look like this:

```
void generatePrognozLenLiteralBlockFilter()
{UBYTE4 freqLL[286]; // array for accumulation of frequencies literal/lengths
memset(freqLL, 0, 286*sizeof(UBYTE4));
unsigned int lenPixellImage, offsetPixellImage, i;
// cycle on the literals of the next layer
for (i=currentPozlImage; i<nextPozStartBlockFilter[indexBlockFilter]; i++)
// if the literal is not a substitute for the nearest processed pixels,
// then accumulate the frequency of the literal after applying the predictor
if (offsetAdjacentPixel[i]==0) freqLL[imageDataPredict[i]]++;
else // otherwise read the replacement and accumulate the frequency of its length
{readReplacelImage(lenPixellImage, offsetPixellImage, i);
freqLL[codesLength[lenPixellImage-3]]++;
i+=lenPixellImage-1; } // move to replace the first pass
memset(lenPrognozLLBlockFilter, 0, 286*sizeof(double));
UBYTE4 count=0;
for (i=0; i<286; i++) count+=freqLL[i]; // the sum of all frequencies of the distribution elements
for (i=0; i<286; i++) // prediction of distribution code lengths
if (freqLL[i]) lenPrognozLLBlockFilter[i]=log2((double)count/freqLL[i]); // by formula (1)
else lenPrognozLLBlockFilter[i]=16; // maximum length
for (i=257; i<286; i++) // add additional bits to the replacement lengths to the format Deflate
lenPrognozLLBlockFilter[i]+=extrasLength[i-PngFirstLengthCode]; }
```

Then the code snippet to check the effectiveness of the next replacement using the predicted code lengths of literals/substitution lengths can be as follows:

```
UBYTE2 lenNDistance=ceil(log2((currentPosImage-current->index+delta)/3.0+
plusOffset))+3; // length of the offset code of the new replacement
deltaReplace=lenPrognozLLBlockFilter[codesLength[countlength-3]]+
lenNDistance; // predicted length of the new replacement code
if (bestlength>0) // was a previous replacement – subtract the length of its code
{// calculate the increase in the length of the replacement
deltaReplace-=lenPrognozLLBlockFilter[codesLength[bestlength-3]];
```

```

// reduce by the length of the offset code of the previous replacement
if (bestPos!=currentPosImage) // the previous replacement was in the dictionary
    deltaReplace=ceil(log2((currentPozImage-bestPoz)/3.0+plusOffset))+3;
else // the previous replacement was for the next processed pixels
    {deltaReplace-=2;
    if (bestoffset>4) deltaReplace--; } // additional offset bits 5 and 6
// predict the length of the code increment of literals
double lenPlusLiteral=0;
bool isAdjacent=false;
// cycle on the growth of literals
for (int i=bestlength; i<countlength && lenPlusLiteral<deltaReplace; i++)
// if the next literal is replaced by the next pixels
if (offsetAdjacentPixel[currentPosImage+i]!=0)
    {readReplacelImage(lenAdjacent, offsetAdjacent, currentPosImage+i);
    isAdjacent=true;
    if (i+lenAdjacent<=countlength) // replacement for the next pixels is included in the new
        lenPlusLiteral+=lenPrognozLLBlockFilter[codesLength[lenSumign-3]]+2;
    i+=lenSumign-1; } // move to replace the first pass
else // otherwise add the predicted length of the literal code
    lenPlusLiteral+=lenPrognozLLBlockFilter[imageDataPredict[currentPosImage+i]];
if (lenPlusLiteral<deltaReplace) // if the literal code is shorter than the replacement code
if (!(bestlength==0 && !isAdjacent)) // first replacements that do not overlap
    // replacements for the next pixels are also included in the LZ77 schedule for analysis
    requiredReplace=false;

```

7. The results of the use of algorithms for rejection of inefficient substitutions in the process of forming the schedule of the modified algorithm LZ77

Let's analyze the results of the use of different variants for rejection of inefficient substitutions in the process of forming the modified algorithm LZ77 (Table 2-3) on the example of progressive hierarchical compression of images of the ACT set. The decoding time of the files obtained as a result of the application of these variants is almost the same, so it was not analyzed here.

We see that the analysis of the increments of additional bits of the offsets reduces the compression ratio on the ACT set for an average of 0.01 bpb due to the discrete-tone images and photorealistic images with the same distant fragments. However, the effect of rejecting inefficient substitutions of the modified schedule of the LZ77 algorithm at the time of coding is not so homogeneous: on average, the ACT coding set has slowed down by 1.11 %, with coding accelerated by 6.7 % for discrete tones and has slow from 1.6 % to 9.7 %.

Table 2

ACT image compression ratios after the application of various options for discarding inefficient substitutions in the process of forming the schedule of the modified algorithm LZ77, bpb

Variants of rejection of inefficient substitutions	№ of the file								Average CR
	1	2	3	4	5	6	7	8	
Without rejection of inefficient substitutions	1.36	0.65	4.65	3.90	4.16	5.20	0.67	4.36	3.12
Analysis of increments of additional bits of offset codes	1.36	0.64	4.65	3.89	4.16	5.18	0.66	4.35	3.11
Elimination of overlays for replacements on the nearest processed pixels	1.34	0.59	4.65	3.82	4.15	5.17	0.62	4.33	3.08
Elimination of overlays and the analysis of devices of additional offset bits	1.34	0.59	4.65	3.82	4.15	5.16	0.62	4.33	3.08
With calculation of the predicted entropy code lengths	1.34	0.57	4.65	3.82	4.15	5.16	0.60	4.33	3.08

Table 3

The time of encoding the images of the ACT set with the use of different variants of rejecting of inefficient substitutions in the process of forming the schedule of the modified algorithm LZ77, c

Variants of rejection of inefficient substitutions	№ of the file								Average time
	1	2	3	4	5	6	7	8	
Without rejection of inefficient substitutions	2.13	3.27	1.03	1.85	1.09	1.86	1.34	1.84	1.80
Elimination of overlays and / or analysis of increments of additional bits	2.19	3.05	1.06	1.93	1.16	2.04	1.25	1.87	1.82
With calculation of the predicted entropy code lengths	2.26	3.85	1.09	2.01	1.18	2.08	1.61	1.87	1.99

This is due to the previous rejection of inefficient substitutions, which allows the discrete-toned images to leave more effective replacements and increases the number of short substitutions that are being found for the next pixels of hierarchical bypass.

The best compression rates for the ACT set are achieved due to the elimination of overlays of dictionary replacements for replacements of the nearest previously processed elements: almost at the same time, the coding of the CR on this set has decreased by an average of 0.04 bpb. At the same time, the improvement of CR is observed for 88 % of the set files, but it is the most significant for discrete-tone images (maximum 0.06 bpb) and photorealistic images with the same distant fragments (maximum 0.08 bpb).

The expected total reduction of CR from the combination of these two options for rejecting of inefficient replacements did not occur (penultimate line of Table 2), because many inefficient substitutions from the dictionary due to increments of additional bits of offset codes are superimposed on replacements on the next processed pixels. Therefore, throughout the ACT set, the maximum reduction in CR from such a combination of 0.01 bpb is observed only for the Sail.bmp.

A significant reduction of image CR is achieved by rejecting of inefficient substitutions with the use of the predicted entropy code lengths: for discrete-toned images, such a reduction is 0.02 bpb (for example, for frymire.bmp it is 9 KB). However, the coding time of images has increased by an average of 9.3% (last row of Table 3) due to the calculation of predicted entropy code lengths for the ACT set, which is unacceptable for standard modes of storing files in graphic formats with such a slight reduction in CR. Therefore, the rejection of inefficient substitutions with the use of the predicted entropy code lengths should be used in archivers and to ensure maximum compression of discrete-toned images and in the standard mode of storing images in format with the implementation of progressive hierarchical compression should be limited with the eliminating overlays of dictionary substitutions for substitutions and with the analysis of increments of additional offset bits.

8. Discussions

Our further research found that the lengths of substitutions of the LZ77 algorithm, multiples of whole pixels, are much more common than the lengths of substitutions that are not multiples of them. And this is not surprising, because pixels of the same color are found in images much more often than the same pixels with equal adjacent components. Replacement lengths of up to ten components correspond to different base values in the literal / replacement length distribution. Therefore, truncating such substitutions to whole pixel lengths (3, 6, or 9 components) by increasing subsequent substitutions should predictably increase the probability of their occurrence and therefore reduce the code lengths. But such truncations reduce the size of not all encoded images, because truncating a sequence of short substitutions can increase their number and therefore increase the overall length of the code. Therefore, our current research is aimed at establishing universal rules for such cuts.

Further, in order to additionally reduce the file size of compressed images without losses and speeding up the decoding in the process of progressive hierarchical compression, we plan to improve the algorithm for reducing the size of the formed Deflate-blocks using entropy code lengths, adapt to such compression other context-sensitive compression methods [13, 14] and increase the efficiency of symmetric and asymmetric predictors using differential color models [15].

9. Conclusions

1. Reducing the size of the compressed images considered by the modified dictionary algorithm LZ77 in the process of progressive hierarchical traversal is achieved mainly on the last layers, because the pixels that are used with the next buffer pixels have on average the highest level of correlation relative to previous layers.

2. It is possible to increase the efficiency of the classical LZ77 algorithm in the process of progressive hierarchical compression without losses by additional search for the same sequences starting from the nearest previously processed pixels.

3. Vocabulary algorithms significantly reduce the CR of primarily artificial discrete-tone images, because such images contain many similar adjacent sequences of brightness of the pixel components.

4. Rejection of inefficient substitutions with the use of the predicted entropy code lengths should be used in archivers and to ensure maximum compression of discrete-toned images, and in the standard mode of storing images in format with the implementation of progressive hierarchical compression should be limited to eliminating overlays of dictionary substitutions for replacements and analysis of increments of additional offset bits.

10. References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, Vol. 1, Dialektika, Kiyv, 2020, 648 p.
- [2] D. Vatolin, A. Ratushnyak, M. Smirnov, Yu. Yukin, Methods of compression of data. device of archivings, compression of images and video, DIALOG-MIFI, Moskow, 2003, 384 p.
- [3] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory, Vol. 23(3) (1977) 337-343.
- [4] P. Deutsch, DEFLATE Compressed Data Format Specification version 1.3, RFC 1951, 1996, Alladin enterprises, May 1996, 15 p.
- [5] D. Huffman, A Method for the Construction of Minimum Redundancy Codes, Proceedings of the IRE, Vol. 40(9) (Sept. 1952) 1098–1101.
- [6] A. Shportko, Use of predictors in the process of progressive hierarchical context-independent lossless image compression, Proceedings of the National University "Lviv Polytechnic". Series: Information Systems and Networks, 771 (2013) 354-364.
- [7] A. V. Shportko, V. A. Postolatii, Development of Predictors to Increase the Efficiency of Progressive Hierarchic Context-Independent Compression of Images Without Losses, Computational Linguistics and Intelligent Systems (COLINS 2021) : Proceedings of the 5th International Conference (Kharkiv, Ukraine, 22-23 April 2021), Kharkiv, Vol. 1. (2021) 1026-1038. URL: <http://ceur-ws.org/Vol-2870/paper77.pdf>.
- [8] A. V. Shportko, Optimizing use of a modified format dictionary compression Deflate during progressive hierarchical lossless image compression, Scientific bulletin of Chernivtsi university. Series: Computer systems and components, Vol. 4 (4) (2013) 40-52.
- [9] ACT – Test Files, 2002. URL: <http://www.compression.ca/act/act-files.html>.
- [10] R. Gonsales, R. Vuds, Digital processing of images, Tekhnosfera, Moskow, 2005, 1072 p.
- [11] H. D. Kothal, M. Tummanapally, V. K. Upadhyay, Review on Lossless Compression Techniques, International conference on computer vision and machine learning, J. Phys.: Conf., Ser. vol. 1228,012007 (2019). doi: 10.1088/1742-6596/1228/1/012007.
- [12] D. Selomon, Compression of data, images and sound, Tekhnosfera, Moskow, 2006, 368 p., Series: World of programming: digital signal processing.
- [13] B. Rusyn, O. Lutsyk, Y. Lysak, A. Lukenyuk, L. Pohreliuk, Lossless image compression in the remote sensing applications, 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP), Lviv, Ukraine (2016) 195-198. doi: 10.1109/DSMP.2016.7583539.
- [14] C. Raghavendra, S. Sivasubramanian, A. Kumaravel, Improved image compression using effective lossless compression technique, Cluster Comput 22 (2019) 3911–3916. <https://doi.org/10.1007/s10586-018-2508-1>.
- [15] A. Shportko, The use of differences of colors models for compression RGB-images without losses, Selection and treatment of information 31 (2009) 90-97.