Dictionary-based compression of map images

Alexandre Akimov

21.12.2001

University of Joensuu
Department of Computer Science
Master´s Thesis

Table of contents

1. Introduction

The map images are considered from the point of view of personal navigation. The main goal of that is to have the maps available in real-time and independent of the location of the user, and without excessive computing resources, because typical navigation devices have limited memory resources and very narrow wireless communication channel. That is why the maps must be compressed before they are transmitted. This means that from this point of view the first goal of maps compression is compression ratio (as the compression could be preformed by powerful servers), and the first goal of decompression is speed and low requirements for machine-resources. Because of this the *block-segmentation* is proposed. The block segmentation divides a whole map into several non-overlapping rectangular parts with the same size, and the blocks are coded separately. It allows transmission of encoded maps by small parts: by encoded blocks.

The main topic of the current thesis is *dictionary-based* methods. The dictionary-based algorithms are compression methods, which are based on the idea of replacing the sequences of literals or pixels by an index or by a pointer to a dictionary, where the dictionary presents a collection of different often-used phrases, which are sequences of literals. And after that, we have got compression because of the replacing. In the case of map images compression dictionary-based methods can also be applied.

In the case of map images we have the situation, where the image subordinates to some logic and because of this some sequences of pixels (as we are dealing with an image, not with text), which also could organize a dictionary. Even if this logic is not clear then it is possible to find out commonly used sequences of pixels, which will became a basement of the dictionary. We can apply for the dictionary elements some *coding* (*Shannon-Fano, Huffman* or *arithmetic*) algorithms, which consist of setting to dictionary indexes *codes*: a short codes to widely used elements and long codes to seldom used. The quality of the compression depends on the dictionary that is why it is so important to find out the optimal way of creating of the dictionary. In the current research there were considered three different ways: static, semi-adaptive and adaptive. Each of the variants has its own achievements and defects.

So the question is how to perform such compression. The semi-adaptive dictionary based compression looks promised: it creates a dictionary during preprocessing stage (because of it the dictionary is adapted to map image), transmit the dictionary into the compressed file and, because of that, the decompression process is little demanding of resources (as does not need to operate with any kind of dictionary-creation process and has only to replace indexes in the compressed file by the codewords from the dictionary.

The rest of the thesis is organized as follows: in Chapter 2 we consider the basics of the *modelling* in the data compression process. There are considered *statistical modelling, context modelling, state* and so on. Chapter 3 describes the some *coding algorithms,* such as *Shannon-Fano, Huffman, and arithmetic* algorithms. In Chapter 4 we talk about the *dictionary-based* compression schemes and *parsing strategies*. Chapter 5 and Chapter 6 are devoted to widely known *LZ77* and *LZ78* algorithms consequently. There are considered principals of these algorithms and ways of their modification. In Chapter 7 we consider the basics of *personal navigation* and *digital map representation* in *raster* format. Chapter 8 describes the semi-adaptive LZW algorithm. Results of the series of experiments, passed over this algorithm, are placed in the Chapter 9.

2. Modelling in data compression

Like in any other scientific or engineering discipline, data compression has a terminology that at first seem overwhelmingly strange to an outsider. To prevent misunderstanding let us take attention to some basic terms of this field of information technology.

In general, data compression consists of taking a stream of symbols and transforming them into codes. If the compression is effective, the resulting stream of codes will be smaller than the stream of original symbols. The decision to output a certain code for a certain symbol or set of symbols is based on a model. The model is simply a collection of data and rules used to process input symbols and determine which code(s) to output. A program uses the model to accurately define the probabilities for each symbol and the coder to produce an appropriate code based on those probabilities. Shortly saying data compression is a modelling plus coding. Modelling and coding are different things. Model applied probabilities to symbols and encoder replaces those symbols by sequences of bits in the output stream.

2.1 Statistical modelling and entropy

Data compression is based on the following abstraction [1]:

$$\text{Data} = \textit{information content} + \textit{redundancy} \tag{2.1}$$

The aim of compression is to remove redundancy and describe the data by its information content. In *statistical modelling* the idea is to "predict" symbols that are to be coded by using a *probability distribution* for the *source alphabet.* Its *entropy* determines the information content of a symbol in the alphabet [2]

$$H(x) = -\log_2 p(x) \tag{2.2}$$

where $p(x)$ is a probability of a symbol. The higher the probability, the lower the entropy, and thus the shorter codeword should be assigned to the symbol. The entropy $H(x)$ gives the number of bits required to code the symbol x in average, on order to achieve the optimal result. The overall *entropy of the probability distribution* is given by [3]

$$H(x) = -\sum_{x=1}^{k} p(x) \cdot \log_2(x) \tag{2.3}$$

where k is the number of symbol in the alphabet. The entropy gives the lower bound of compression that can be achieved (measured in bits per symbol), corresponding to the model. The optimal compression can be realized by *arithmetic coding* [14].

All modelling schemes can be classified into three following categories [2]:

- *Static* modelling
- *Semi-adaptive* modelling
- *Adaptive* (or *dynamic*) modelling

Static modelling is the simplest form of statistical modelling. In the earliest days of information theory, the CPU cost of analysing data was considered significant, so data analysis wasn't frequently performed and static models were used instead [1]. In the static modelling the same code table is applied to all input data ever to be coded (see Figure 2-1). Consider text

compression; if the ASCII data to be compressed is known to consist of English text, the model based on the frequency of English text are $p('e') = 10\%$, $p('t') = 8\%$ on average for this two letters. Unfortunately, defect of such modelling is that such scheme will give bad compression each time, when the coding data do not coincide with chosen probability model, so static modelling is used only when the most important things in compression are speed and simplicity of realization.

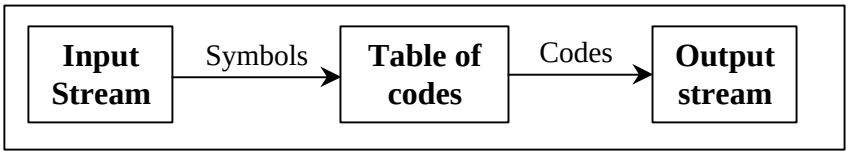


Figure 2-1: Principal scheme of static modelling: table of codes does not depend from input stream.

Semi-adaptive modelling solves this problem by using for each data stream its own probability model, which is created before the compression by preliminary analysis of the data. So, the model has to be transferred to decoder before beginning of the compression. In spite of these additional expenditures on the creating and transferring of the model, this model achieves the best compression results. Scheme of semi-adaptive modelling is shown in Figure 2-2.

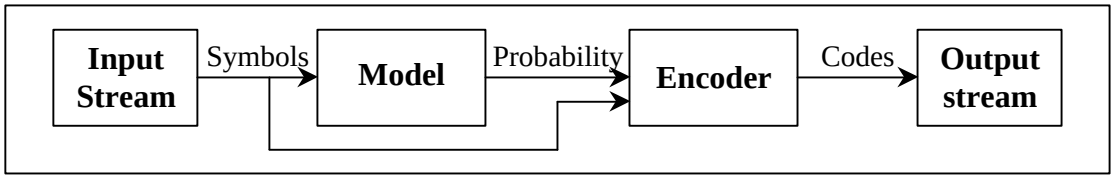


Figure 2-2 Scheme of semi-adaptive modelling: the model is created before the compression stage, and during the compression, created code is based on the symbol and probability, given by model.

Adaptive or dynamic modelling is quite elegant and effective way to create the model: the coding process during one phase (instead two phases in the semi-adaptive modelling) and the statistical model is created “on-line” during compression. The probabilities of each symbol can be calculated adaptively with the help of array of counts: one count for each symbol, in the beginning each count has a value one, and after encoding of a symbol the value of the count is increased by one. Analogically during decoding the decoder increases the value of symbol’s count by one. The probability of each symbol is defined by its relative frequency. The compressor encodes a symbol using the existing model, and then it updates the model to account for the new symbol (see Figure 2-3). The decompressor likewise decodes a symbol using the existing model, and then it updates the model (see Figure 2-4). As long as the algorithm to update the model operates identically for the compressor and the decompressor, the process can operate perfectly without needing to pass a statistics table from the compressor to the decompressor. Adaptive data compression has a slight disadvantage in that it starts compressing with sub optimal statistics. By subtracting the cost of transmitting the statistics with the compressed data, however, an adaptive algorithm will usually perform better than a fixed statistical model.

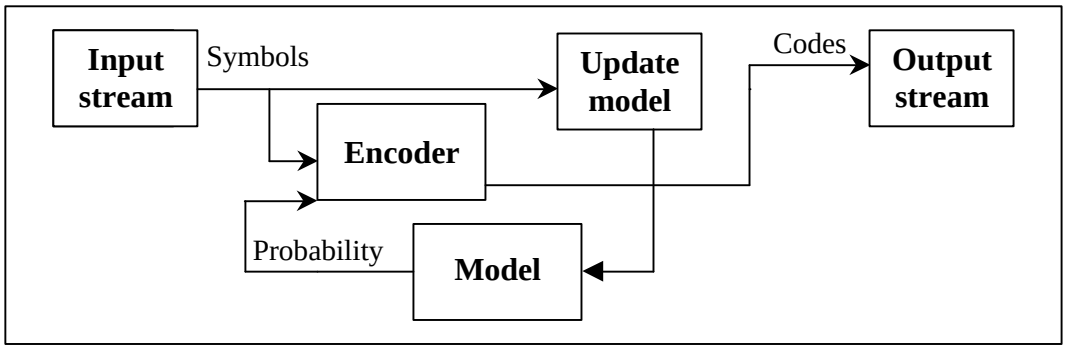


Figure 2-3 Scheme of the adaptive modelling: the model sends to encoder the symbol. The symbol is coded by old model, and then the model is updating itself.

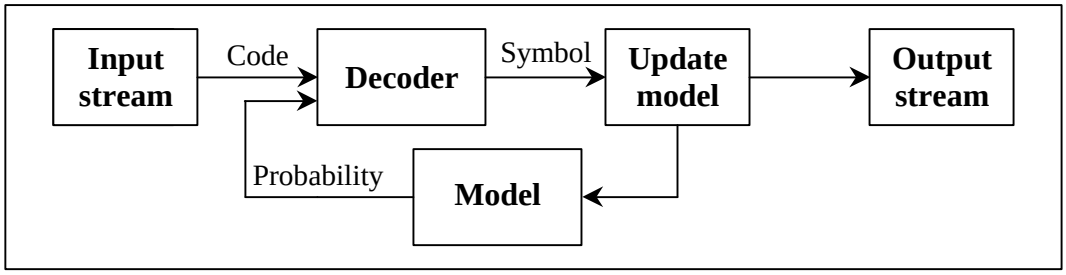


Figure 2-4 Scheme of adaptive decompression: an existing model decodes a code from input stream, and after it the model is updated.

Consider the sequence "*a, a, a, b, b, a, b, c, c, a*". If no priory knowledge is available the static modelling is a simple model that could be applied to the compression process. The compression process with the static modelling is shown in Table 2-1. In Tables 2-2 and 2-3 are represented semi-adaptive and dynamic modelling consequently.

Table 2.1: The example of the static modelling. The overall entropy of probability distribution is $H = \frac{(1.58+1.58+1.58)}{3} = 1.58$

Static model		
symbol	$p(x)$	$H(x)$
A	0.33	1.58
B	0.33	1.58
C	0.33	1.58

Table 2-2: The example of the semi-adaptive modelling. $H = 0.5 \cdot 1 + 0.3 \cdot 1.74 + 0.2 \cdot 2.32 = 1.49$

Semi-adaptive model			
Symbol	**Count**	$p(x)$	$H(x)$
A	**5**	0.5	1
B	**4**	0.3	1.74
C	**2**	0.2	2.32

In the case of adaptive modelling the input data is processed as shown in Table 2-3. In the beginning of the process some initial model is needed since no prior knowledge is allowed from the input data. Here we assume equal probabilities. The probability of the first symbol (here 'a')

is 0.33 and the corresponding entropy is 1.58. After that the model is updated by increasing the count of 'a' by one. Note, that it is assumed that that each symbol has occurred once before the proceeding so their initial counts equal to one. This is made to avoid so-called *zero-frequency problem* [2]. If we have no occurrence of a symbol, its probability would be 0 that yields entropy to infinity.

Table 2-3: The example of adaptive modelling: the numbers are counts of symbols on each step.

	step									
symbol	*1.*	*2.*	*3.*	*4.*	*5.*	*6.*	*7.*	*8.*	*9.*	*10.*
a	**1**	**2**	**3**	3	3	**4**	4	4	4	**5**
b	1	1	1	**1**	**2**	2	**3**	3	3	3
c	1	1	1	1	1	1	1	**1**	**2**	2
$p(x)$	0.33	0.50	0.60	0.20	0.22	0.57	0.38	0.125	0.22	0.5
H	1.58	1	0.74	2.32	2.19	0.81	1.4	3	2.19	1

The corresponding entropies of the different modelling strategies are summarized here:

Static modelling:	1.58 (bits per symbol)
Semi-adaptive modelling:	1.49
Dynamic modelling	1.62

The properties of the three different modelling strategies are summarized as follows [3]:

Static modelling	**Semi-adaptive modelling**	**Dynamic modelling**
+One-pass method	-Two-pass method	+One-pass method
+No side information	-Side information needed	+No side information
-Non-adaptive	+Adaptive	+Adaptive
+No updating of model during compression	+No updating of model during compression	-Updating of model during compression

2.2 Context models

So far we have considered the statistical with the overall frequency distribution of the source, but paid no attention to the spatial dependencies between the individual symbols.

Once again, for example in English text the probabilities of the symbols ".", "e", "t", "k" usually are 18%, 10%, 8%, 0.5% [1]. Consequently the corresponding entropies are 2.47, 3.32, 3.64 and 7.62. So we can code the symbols by 4.5 bits on average. This is a simple context-free model.

More complicated way of calculating of probabilities is in defining of dependences of symbols from the previous symbols. For example: the probability of the case that the letter "u" will appear after letter "q" in English is about 99% [1], whereas without dependency from the previous symbol is about 2.4% [1]. So with taking into account of the context the entropy of the letter "u" is about 0.014 and otherwise about 5.38.

This type of models is possible to generalize concerning to previous symbols, that are chosen to define the probability of the next symbol. In generally it calls *finite-context* modelling. The *order* of the model refers to the number of previous symbols that make up the context. The simplest finite-context model would be an *order-0* model that calculates the probability of each symbol independently of any previous symbols. To implement this model, all we need is a table containing the frequency counts for each symbol that might be found in the input stream. For

example the alphabet of the input stream consists of 256 different elements, (256 is the number of different symbols in ASCII) then an *order-1* model stores track of 256 different tables of frequencies. Similarly, an *order-2* model needs to handle 65,536 (256^2) different tables of contexts [1] (see Table 2-4). The entropy of an *order-n* context model is the weighted sum of the entropies of the individual contexts:

$$H_n = -\sum_{j=1}^{n}\left[\sum_{i=1}^{k} p(x_i|c_j)\cdot \log_2\left(p(x_i|c_j)\right)\right]. \tag{2.4}$$

Table 2-4: An order-2 context table for input "ABCABDABE"

order-0		
Context	**Symbol**	**Count**
""	A	3
""	B	3
""	C	1
""	D	1
""	E	1
order-1		
"A"	B	3
"B"	C	1
"B"	D	1
"B"	E	1
"C"	A	1
"D"	A	1
order-2		
"AB"	C	1
"AB"	D	1
"AB"	E	1
"BC"	A	1
"CA"	B	1
"BD"	A	1

Here is shown that even so low-level context model with so small alphabet can create a big table of contexts. That is the reason why adaptive schemes of modelling are used with contexts, because in the case of the semi-adaptive model we need to store the statistics in the compressed file.

For optimal context choosing such scheme called *blended context* [2], can be applied. It uses big order contexts for better compression and small order contexts in insufficiently good, and the probability estimation is based on the contexts with different length. These schemes of context modelling were described in [2]. The way to unite all order contexts is to set to each order each own weight.

Let's assume $p(o, j)$ is a probability of symbol j from the alphabet A in the finite-context model with order o**.** This probability was set adaptively and will change during compression process. Let's assume $w(o)$ as the weight of the model with order o, and the maximum context order is m, so the blended probabilities $p(j)$ are:

$$p(j) = \sum_{o=-1}^{m} w(o) \cdot p(o, j), \tag{2.5}$$

$$\text{where} \quad \sum_{o=-1}^{m} w(o) = 1. \tag{2.6}$$

(Condition 2.5 is not obligatory, but it is used to not exceed the limit of the value that was used by arithmetic coder). The counters are used to calculate the values of the probabilities weights, which are used quite often and related to each context. Let's assume $c(o, j)$ as the number of appearance of the symbol j in the current context with order o**,** and let $C(o)$ be the overall number of the context appearance. Then is:

$$C(o) = \sum_{j \in A} c(o, j) \tag{2.7}$$

A simple *blending* can be constructed from the mechanism of choosing the estimation of the context:

$$p(o, j) = \frac{c(o, j)}{C(o)} \tag{2.8}$$

The probabilities of the symbols from previous example with static weights 0.7, 0.2, 0.1 for models with *order-2*, *order-1*, *order-0* consequently are shown below.

Symbol "A": $0.33 \cdot 0.7 + 0.25 \cdot 0.2 + 0.33 \cdot 0.1 = 0.314$,
symbol "B": $0.167 \cdot 0.7 + 0.375 \cdot 0.2 + 0.33 \cdot 0.1 = 0.225$,
symbol "C": $0.167 \cdot 0.7 + 0.125 \cdot 0.2 + 0.11 \cdot 0.1 = 0.153$,
symbol "D": $0.167 \cdot 0.7 + 0.125 \cdot 0.2 + 0.11 \cdot 0.1 = 0.153$,
symbol "E": $0.167 \cdot 0.7 + 0.125 \cdot 0.2 + 0.11 \cdot 0.1 = 0.153$.

The *zero frequency-problem* appears when there is no information about the input, so we haven't seen some symbols in this context yet, and the problem is which probability distribution to choose. There are at least two ways to solve this problem.

- The first one is to use the *flat*, or equal, distribution of probabilities of the symbols. This method is not quite effective, because for example if we have a symbol, that has not been observed before in the given context, its *flat* probability distribution is 1/256 (or 1/*Z*, where *Z* is the number of symbols in the alphabet), which gives us the same probability, as to every new symbol. But in this case, after the next appearance of this symbol its probability will be 2/257, after third appearance its probability will be 3/258, and so on. But in the same time the real probability of the symbol can be about 2/3.
- Another way is to use an *escape code*, which means that the symbol, which is currently coding is not in the current context, and thus the model *escapes* to another context. The model starts in the *order-0* table with this special *escape* symbol, which has probability of 1, and with all alphabet symbols which have zero probability. The *escape code* is used to make decoder know that it has to use next table, and the model always goes from high orders to lower order, until it will fall into *order-(-1)*, where it can code the symbol in case it hasn't appeared in higher orders. For example if we have to process a particular symbol with *order-(x)*, and this symbol hasn't been observed with this order earlier, the model will decrease the order, and try to find the match and process this symbol, until it will fall in *order-(-1)* model where all symbols from the alphabet have *flat* probability, and thus can safely process the symbol.

The probability of the *escape code* (*escape probability*), is defined as *e(o)* (where *o* is the order of the context). In the blended context models *escape probability* is equal to the *extent of importance* of the *order-(o)*, because it defines how the model will continue to work with *order-(o)* context or will escape to lower-order contexts. From the other hand the role of the *extent of importance* in the blended context modelling is playing by the weights, applied to different context. We can estimate different weights as:

$$w(o) = (1 - e(o)) \cdot \prod_{i=o+1}^{m} e(o), \text{ for } -1 \le o < m, \tag{2.9}$$

$$w(m) = (1 - e(m)), \tag{2.10}$$

where *m* is the maximum order in the blended context model. In other words, the weights, they are the probability that model will escape to the *order-**o*** and will not escape to lower orders. The main achievements of this mechanism of weights defining is that it is adaptive and takes into account the importance of the changing of different order context's.

The *escape probability* is the probability that the *zero-frequency problem* will appear. In theory there is no basis for optimal defining of the escape probability [2]. Some ways of the probability definition are described below.

- The first one is to have an additional counter, which will count the appearance of the new symbols. The formula of the *escape probability* is shown below [2].

$$e(o) = \frac{1}{C(o) + 1}, \tag{2.11}$$

  where $C(o)$ is the overall number of appearance of the given context.
- The second one is based on following abstraction "We will not believe in this, until we will see it twice", It works by subtracting one from the every symbol counters, and do not give an estimation of the symbol until it will appear at least the second time. The escape probability in this case is:

$$e(o) = \frac{q(o)}{C(o)}, \tag{2.12}$$

  where *q(o)* is the number of different symbols in the order-o context.
- In the third method the escape probability is equal to the number of all different symbols seen so far. (with the escape probability, added to overall probability) [2]:

$$e(o) = \frac{q(o)}{C(o) + q(o)}, \tag{2.13}$$

The context modelling gives us the best compression from all known statistical models, but the coding and decoding process can be very slow [2]. According to any practical scheme of its implementation the time of coding and decoding is increasing only linear with increasing of the text's size. Beside this the processing time can be increased at least linearly due to the order of the context model.

2.3 State models

The *statistical models* with final number of the states are based on final automaton. They have a collection of states *S(i)* and probabilities of the model transition between the states *P(i, j)*, from the state *i* to state *j*. And each transition is defined by unique symbol. That is why any source text will assign unique way in models (if it exists) through a sequence of such symbols. Often such models are named as Markov's models though sometimes this term is inexact used for indication finite-context models.

The models with final number of the states are able to imitate finite-context models. For example, *order-0* context model of simple English text has one state with 27 transitions back to this state: 26 for letters and 1 for space. The order-1 context model has 27 states, each with 27 transitions. The order-n context model has 27^n conditions with 27 transitions for each of them.
The entropy of the finite state process with states *S(i)* is simply the average value of the entropy at each state:

$$H(x) = \sum_{i=1}^{M} P(S(i)) \cdot H(S(i)) \tag{2.14}$$

For example consider a binary image. It has only two different states: black pixel or white pixel (S_b and S_w consequently). Transition probabilities are defined as *P(w|b)*: the probability of white pixel is followed by a black pixel, *P(b|w)*: the probability of black pixel is followed by white pixel, *P(w|w)*: for two white pixels; one after another, *P(b|b)*: same, but for black pixels. The model is shown in the Figure 2-5.

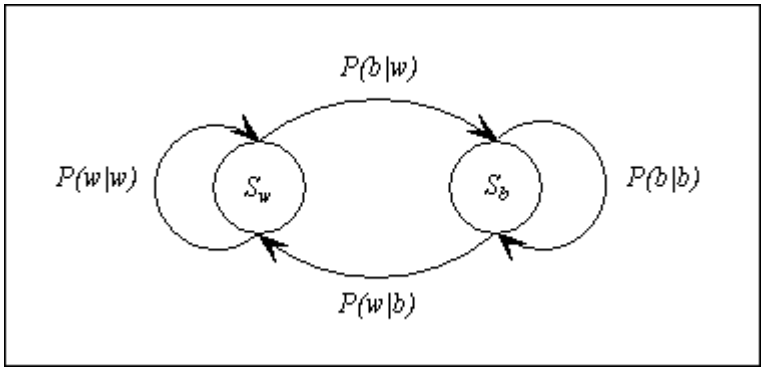


Figure 2-5: Two-state model for binary images.

As far as the number of states is two:

$$H(S_w) = -P(b|w) \cdot \log_2 P(b|w) - P(w|w) \cdot \log_2 (w|w) \tag{2.15}$$

$$H(S_b) = -P(w|b) \cdot \log_2 P(w|b) - P(b|b) \cdot \log_2 (b|b), \tag{2.16}$$

where $P(w|w) = 1 - P(b|w)$.

Assuming that the values of probabilities:

$$P(S_w) = 0.8,\ P(S_b) = 0.2,\ P(w|b) = 0.3,\ P(b|w) = 0.01.$$

Then the entropy of the simple statistical modeling is:

$$H = -0.8 \cdot \log_2 0.8 - 0.2 \cdot \log_2 0.2 = 0.722 \text{ bits}$$

Now in the state model:

$$H(S_b) = -0.3 \cdot \log_2 0.3 - 0.7 \cdot \log_2 0.7 = 0.881 \text{ bits},$$
$$H(S_w) = -0.01 \cdot \log_2 0.01 - 0.99 \cdot \log_2 0.99 = 0.081 \text{ bits}.$$

The result entropy of state model is 0.241, which is about the third part of the entropy, obtained by simple statistical model.

2.4 Move To Front transformation

Move To Front (MTF) is a transformation algorithm that does not compress data but sometimes it can help to reduce redundancy.

Instead of outputting the symbol (byte), MTF algorithm outputs a code, which refers to the position of the symbol in a table with all the symbols, thus the length of the code is the same as the length of the symbol (when using bytes as symbols, byte based output can be performed). Both encoder and decoder should initialise the table with the same symbols in the same positions. Once a symbol is being processed the encoder outputs its position in the table and then put it in the top of the table, position 0. All the codes from the position till the position of the symbol being coded are moved to the next position. This simple scheme assigns codes with lower values for more redundant symbols (symbols which appear more frequently) like the following. Assuming the input string as ”abaacabad”. The coding process is shown in Table 2-5:

Table 2-5: MTF transformation for input string ”abaacabad”.

Symbol	A	B	A	A	C	A	B	A	D
Code	0	1	1	0	2	1	2	1	3
List	ABCD	BACD	ABCD	ABCD	CABD	ACBD	BACD	ABCD	DABC

The entropy for original string is:

$$H = -\frac{5}{9} \cdot \log_2 \frac{5}{9} - \frac{2}{9} \cdot \log_2 \frac{2}{9} - \frac{1}{9} \cdot \log_2 \frac{1}{9} - \frac{1}{9} \cdot \log_2 \frac{1}{9} = 0.471 + 0.482 + 0.352 + 0.352 = 1.633 \text{ bits}.$$

The entropy for transformed string is:

$$H = -\frac{2}{9} \cdot \log_2 \frac{2}{9} - \frac{4}{9} \cdot \log_2 \frac{4}{9} - \frac{2}{9} \cdot \log_2 \frac{2}{9} - \frac{1}{9} \cdot \log_2 \frac{1}{9} = 0.285 + 0.482 + 0.482 + 0.352 = 1.602 \text{ bits}.$$

3. Coding

The task of replacing symbol with probability p approximately by $-\log_2 p$ bits is called *coding* [2]. The coder operates by values of probabilities, which defines the next outputting symbol. The coder creates a stream of bits. The symbol can be uniquely decoded, using this stream, if decoder uses the same probability model.

3.1 Shannon-Fano coding

The first well-known method for effectively coding symbols is now known as Shannon-Fano coding. Claude Shannon at Bell Labs and R.M. Fano at MIT developed this method nearly simultaneously. It depended on simply knowing the probability of each symbol's appearance in a text. Given the probabilities, a table of codes could be constructed that has several important properties:

1. Different codes have different numbers of bits.
2. Codes for symbols with low probabilities have more bits, and codes for symbols with high probabilities have fewer bits.
3. Though the codes are of different bit lengths, they can be uniquely decoded.

The Shannon-Fano algorithm creates a table of codes, which has a tree structure. Decoding an incoming code consists of starting at the root, then turning left or right at each node after reading an incoming bit from the data stream. Eventually, if a leaf of the tree is reached, that means that the appropriate symbol is decoded.

A Shannon-Fano tree is built according to a specification designed to define an effective code table. The actual algorithm is simple:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
2. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the top and the least common at the bottom.
3. Divide the list into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.
4. The upper half of the list is assigned the binary digit 0, and the lower half is assigned the digit 1. This means that the codes for the symbols in the first half will all start with 0, and the codes in the second half will all start with 1.
5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

A simple example of Shannon-Fano algorithm is shown in Table 3-1.

Table 3-1: An example of Shannon-Fano code tree creating process

Symbol	Count	First division	Second division	Third division	Fourth Division
A	15	0	0	—	—
B	7	0	1	—	—
C	6	1	0	—	—
D	6	1	—	1	0
E	5	1	—	1	0

In Table 3-1 five symbols (*A, B, C, D, E*) are processing in the tree construction process. Before starting of divisions all symbols are sorted by their counts. During first division they are separated into two groups "AB" and "CDE". As the first group has two symbols only, the second division is the last one for them. But the second group has three symbols and it needed two divisions to create the code tree. The third division separates the second group into two subgroups: "C" and "DE". The fourth division processes with the second subgroup (as there is nothing to divide in the first subgroup: it has only one symbol). The code 0 is applied to *D* (as *D* has bigger count) and code 1 is applied to *E*. The resulting tree is shown in the Figure 3-1.

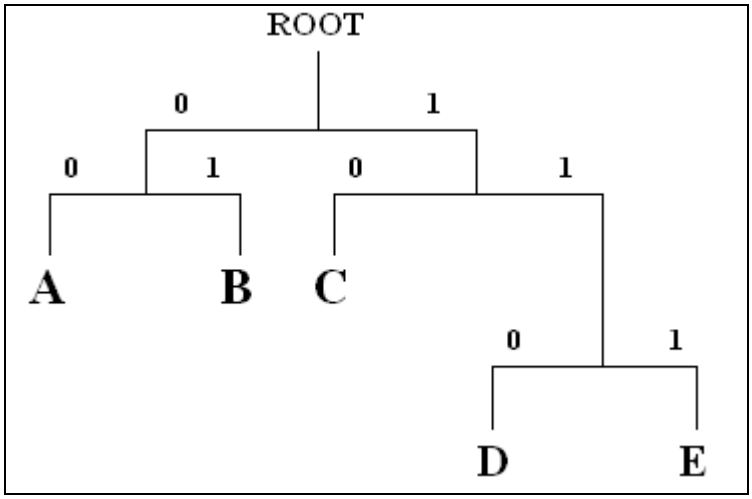


Figure 3-1: The resulting Shannon-Fano tree of the example

The codebook, created during Shannon-Fano algorithm is placed in Table 3-2.

Table 3-2: Comparing SF symbol's code size with the entropy of the symbol.

Symbol	Count	Entropy	Code length	Code
A	15	1.38	2	00
B	7	2.48	2	01
C	6	2.70	2	10
D	6	2.70	3	110
E	5	2.96	3	111

$H = 1.38 \cdot 15 + 2.48 \cdot 7 + 2.70 \cdot 6 + 2.96 \cdot 5 = 85.25$ bits.
$\text{SF size} = 2 \cdot 15 + 2 \cdot 7 + 2 \cdot 6 + 3 \cdot 6 + 3 \cdot 5 = 89.00$ bits.

The main result of this example is that for encoding 85.25 bits of information Shannon-Fano algorithm uses 89 bits.

3.2 Huffman coding

Huffman first published his paper on coding in 1952 [15], and it instantly became the most-cited paper in Information Theory. It probably still is. Huffman's original work spawned numerous minor variations, and it dominated the coding world till the early 1980s [1].

The procedure for building the tree is simple and elegant. The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight, which is simply the frequency or probability of the symbol's appearance. The tree is then built with the following steps [15]:

1. The two free nodes with the lowest weights are located.
2. A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.
3. The parent node is added to the list of free nodes, and the two child nodes are removed from the list.
4. One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.
5. The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.

Analogically to the previous example, during Huffman tree construction process could be used as symbol's probabilities, as symbol's counts. So if there is a list of five symbols (see Figure 4-2a). Two nodes with smallest count ("*D*" and "*E*") are replaced by a node "*DE*" (see Figure 4-2b). The new-created node "*DE*" has count eleven, which is the sum of five and six: counts of "*E*" and "*D*" accordingly. After this the same procedure repeats for nodes "*B*" and "*C*". The resulting node "*BC*" has count 13. Nodes "*BC*" and "*DE*" still have the smallest counts. That is why they are united. There are two nodes only in the list of nodes after this operation. When nodes "*A*" and "*BCDE*" will be united the algorithm is finished. The resulting tree is shown in the Figure 3-2c, and the resulting table of codes is shown in Table 3-4.

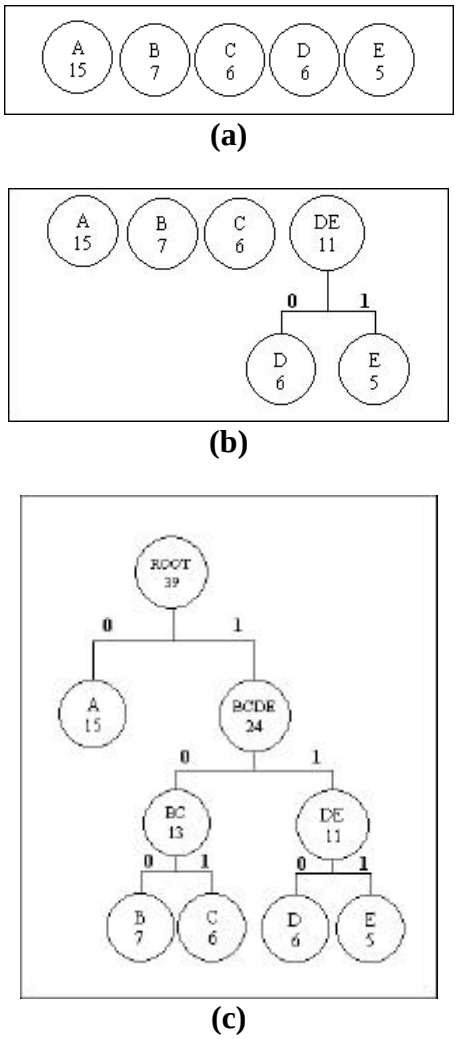


Figure 3-2: Construction of the Huffman tree; (a) leaf nodes; (b) combining nodes; (c) resulting tree.

Table 3-4: The resulting table of codes in Huffman coding process.

Symbol	Code
A	0
B	100
C	101
D	110
E	111

It is clear that Huffman codes differ from Shannon-Fano codes. In the case of Huffman coding it is needed 87 bits to encode a message with 85.25 bits of information content and Shannon-Fano encoding requires 89 bits.

In general, Shannon-Fano and Huffman coding are close in performance. But Huffman coding will always at least equal the efficiency of Shannon-Fano coding, so it has become the predominant coding method of its type. Since both algorithms take a similar amount of processing power, it seems sensible to take the one that gives slightly better performance. And Huffman was able to prove that this coding method cannot be improved on with any other integral bit-width coding stream [1].

3.3 Run-Length encoding

Data files frequently contain the same character repeated many times in a row. For example, text files use multiple spaces to separate sentences, indent paragraphs, format tables and charts, etc. Digitised signals can also have runs of the same value, indicating that the signal is not changing. For instance, an image of the nighttimes sky would contain long runs of the character or characters representing the black background. Likewise, digitised music might have a long run of zeros between songs. Run-length encoding is a simple method of compressing these types of files.

Figure 3-3 illustrates run-length encoding for a data sequence having frequent runs of zeros. Each time a zero is encountered in the input data, two values are written to the output file. The first of these values is a zero, a flag to indicate that run-length compression is beginning. The second value is the number of zeros in the run. If the average run-length is longer than two, compression will take place. On the other hand, many single zeros in the data can make the encoded file larger than the original. Many different run-length schemes have been developed. For example, the input data can be treated as individual bytes, or groups of bytes that represent something more elaborate, such as floating point numbers. Run-length encoding can be used on only one of the characters (as with the zero above), several of the characters, or all of the characters [7].

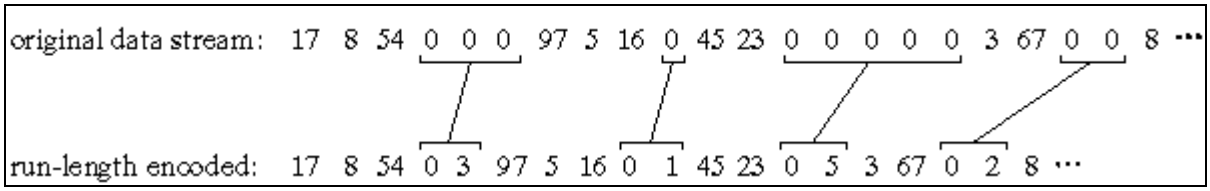


Figure 3-3: An example of run-length encoding, where RLE is used for one symbol; “0”.

3.4 Arithmetic coding

Huffman codes have to be an integral number of bits long, and this can sometimes be a problem. If the probability of a character is 1/3, for example, the optimum number of bits to code that character is around 1.6 bits. Huffman coding has to assign either one or two bits to the code, and either choice leads to a longer compressed message than are theoretically possible.

This non-optimal coding becomes a noticeable problem when the probability of a character is very high. If a statistical method could assign a 90 percent probability to a given character, the optimal code size would be 0.15 bits. The Huffman coding system would probably assign a 1-bit code to the symbol, which is six times longer than necessary.

The idea of *arithmetic coding* is to represent the input file as a single number between the range [0,1]. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. To construct the output number, the symbols are assigned a set of probabilities. The message "*BILL GATES*", for example, would have a probability distribution of symbols like is shown in Table 3-5.

Table 3-5: Probability distribution of symbols in string "*BILL GATES*".

Symbol	Probability
SPACE	0.1
A	0.1
B	0.1
E	0.1.
G	0.1.
I	0.1.
L	0.2.
S	0.1.
T	0.1.

Once symbols probabilities are known, individual symbols need to be assigned a range along a "probability line", nominally from 0 to 1. It does not matter, which symbols are assigned which segment of the range, as long as it is done in the same manner by both the encoder and the decoder. The nine- symbol set used here would look like the in Table 3-6.

Table 3-6: The set of ranges for arithmetic coding in message "*BILL GATES*".

Symbol	Range
SPACE	[0, 0.1)
A	[0.1, 0.2)
B	[0.2, 0.3)
E	[0.3, 0.4)
G	[0.4, 0.5)
I	[0.5, 0.6)
L	[0.6, 0.8)
S	[0.8, 0.9)
T	[0.9, 1)

During the rest of the encoding process, each new symbol will further restrict the possible range of the output number. The next character to be encoded, the letter I, owns the range 0.50 to 0.60

in the new subrange of 0.2 to 0.3. So the new encoded number will fall somewhere in the 50th to 60th percentile of the currently established range. Applying this logic will further restrict the number to 0.25 to 0.26. The algorithm to accomplish this for a message of any length is shown in the Figure 3-4.

```
LowBoundOfInreval ← 0.0
HighBoundOfInterval ← 0.0
While(Not end of stream)
{
  c ← ReadSymbol(InputStream)
  RangeOfInterval ← HighBoundOfInterval – LowBoundOfInreval
  HighBoundOfInreval ← LowBoundOfInreval + RangeOfInterval*HighRangeOfSymbol(c)
  LowBoundOfInreval ← LowBoundOfInreval + RangeOfInterval*LowRangeOfSymbol(c)
}
Output(LowBoundOfInterval)
```

Figure 3-4: The encoding process in arithmetic coding.

Table 3-7: The result of arithmetic encoding of the message ”*BILL GATES*”.

New Character	Low value of the range	High value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

So the final low value, 0.2572167752, will uniquely encode the message “*BILL GATES*” using our present coding scheme.

Given this encoding scheme, it is relatively easy to see how the decoding process operates. Find the first symbol in the message by seeing which symbol owns the space our encoded message falls in. Since 0.2572167752 falls between 0.2 and 0.3, the first symbol must be *B*. Then remove *B* from the encoded number. Since it is know that the low and high ranges of B, remove their effects by reversing the process that put them in. First, subtract the low value of *B*, giving 0.0572167752. Then divide by the width of the range of *B* by 0.1. This gives a value of 0.572167752. Then calculate where that lands, which is in the range of the next letter, *I*. The algorithm for decoding the incoming number is shown in the Figure 3-5. The decoding algorithm for the “*BILL GATES*” message will proceed as shown in Table 3-8.

```
LowBoundOfInterval ← ReadCode (InputStream)
While (LowBoundOfInreval > 0)
{
  Symbol ← FindSymbolStraidingThisRange(LowBoundOfInterval)
  Output(Symbol)
  RangeOfInterval ← HighRangeOfSymbol(Symbol)-LowRangeOfSymbol(Symbol)
  LowBoundOfInterval ← LowBoundOfInterval-RangeOfInterval*LowRangeOfSymbol(Symbol)
}
```

Figure 3-5: Brief description of arithmetic decoding process.

Table 3-8: The decoding process for message "*BILL GATES*".

Encoded Number	Output Symbol	Low bound	High bound	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

Encoding and decoding a stream of symbols using arithmetic coding is not too complicated. But at first glance it seems completely impractical. Most computers support floating-point numbers of around 80 bits. Is it necessary to start algorithm again after encoding of fifteen symbols? As it turns out, arithmetic coding is best accomplished using standard 16-bit and 32-bit integer math. Floating-point math is neither required nor helpful [1].

At the first time it is necessary to consider the fundamental properties of binary arithmetic. With n bits at most 2^n different combinations can be represented; or with n bits a code interval between zero and one can be divided into 2^n parts each having the length of 2^{-n}, see Figure 3-6. If A is a power of 0.5, that is $A = 2^{-n}$. From the opposite point of view, an interval with the length A can be coded by using $-\log_2 A$ bits (assuming A is a power of 2).

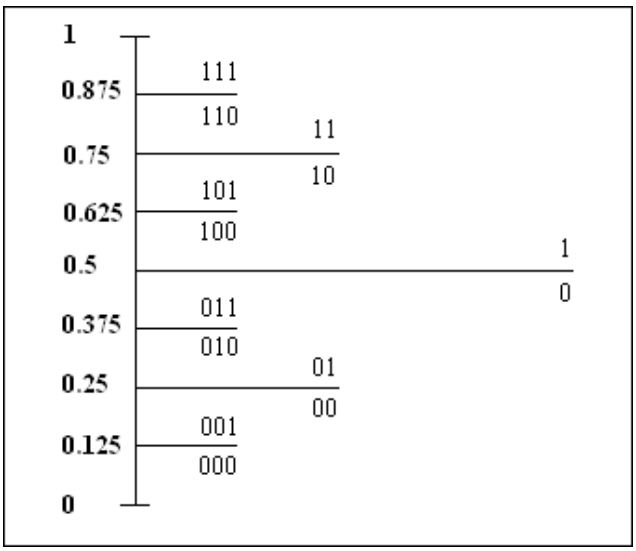


Figure 3-6: Interval [0,1] is divided into 8 parts, thus each having the length of $2^{-3} = 0.125$ (Each interval can be now coded by using $-\log_2 0.125 = 3$ bits).

And as the in the final of the process is an interval (the range of the interval and its low bound are known), it can be coded by the same method.

During implementation the final interval can gets so small that it cannot be expressed by 16 bit or 32 bit integer in computer memory. The following procedure is thus implied. When the interval falls completely below 0.5 (the half point of the interval [0,1]), it is known that the codeword, describing the final interval, starts with the bit 0. If the interval were above the half point, the codeword would start from the bit 1. In both cases, the starting bit can be outputted and the processing can be limited to the corresponding, which is either [0,0.5], or [0.5,1]. This is realized by zooming the corresponding half as shown in the Figure 3-7.

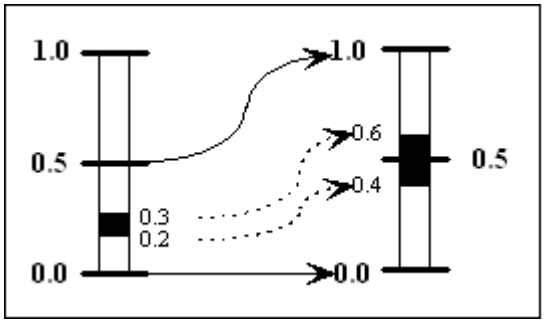


Figure 3-7: Example of *half-point zooming.*

The underflow can also occur if the interval decreases so that its lower bound is just below the half point, but the upper bound is still above. In this case the half-point zooming cannot be applied. The solution is so-called *quarter-point zooming*, see Figure 3-8. The condition for quarter-point zooming is that the lower bound of the interval exceeds the 0.25 and the upper bound does not exceed 0.75. Now it is known that the output bit stream is either “01xxx” or “10xxx” if the final variant is above the half point (Here xxx defines the rest of the code stream).

Since the final interval completely covers either the range [0.25,0.5] or the range [0.5,0.75] the encoding can be finished by sending the bit pair “01” if the upper bound is below 0.75, or “10” if the lower bound exceeds 0.25.

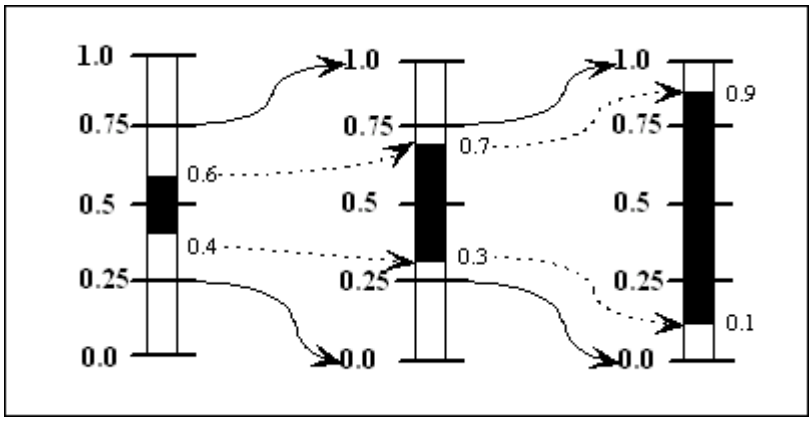


Figure 3-8: Example of two subsequent quarter-point zooming.

4. Dictionary based schemes

So far, the compression methods described before used a statistical model to encode single symbols. They achieve compression by encoding symbols into bit strings that use fewer bits than the original symbols. The quality of the compression goes up or down depending on how good the program is at developing a model. The model not only has to accurately predict the probabilities of symbols, it also has to predict probabilities that deviate from the mean. More deviation achieves better compression.

But dictionary-based compression algorithms use a completely different method to compress data. This family of algorithms does not encode single symbols as variable-length bit strings; it encodes variable-length strings of symbols as single tokens. The tokens form an index to a phrase dictionary. If the tokens are smaller than the phrases they replace, compression occurs. In many respects, dictionary-based compression is easier for people to understand. It represents a strategy that programmers are familiar with: using indexes into databases to retrieve large amounts of storage. An example of dictionary-based coding is shown in the Figure 4-1. Here is a dictionary of names. Each name has its own index. A word from the input string is found ("Bill") and is replaced by its index.

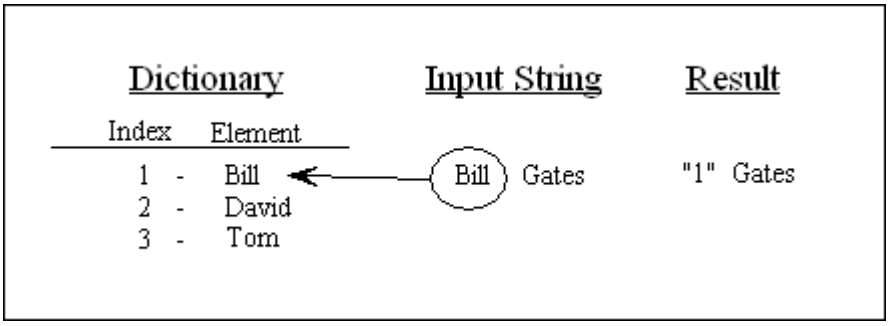


Figure 4-1: An example of dictionary-based coding.

4.1 Static dictionaries

In some cases, it is advantageous to use a predefined dictionary to encode text. If the text to be encoded is a database containing all motor-vehicle registrations for Texas, we could develop a dictionary with only a few thousand entries that concentrated on words like "Mercedes", "Akimov", "Main", and "1977". Once this dictionary was compiled, it could be kept on-line and used by both the encoder and decoder as needed.

A dictionary like this is called a *static dictionary*. It is built up before compression occurs, and it does not change while the data is being compressed. The example in the Figure 4-1 is an example of the static dictionary scheme.

One fast algorithm that has been proposed several times in different forms is diagram coding, which maintains a dictionary of commonly used diagrams, or pairs of characters. At each coding step the next two characters are coded together, otherwise only the first character is coded. The coding position is shifted by one or two characters as appropriate [2].

In general the diagram coding can be described in next way: some characters (let us define the number of characters as q) are considered as *essential*. Then is taken a cross-product of the set of characters with itself. The result is a dictionary.

4.2 Semi-adaptive dictionaries

The logical developing of diads (the sequence of symbols with the length 2) coding idea is creating for each text each own dictionary. Such way of coding is called *semi-adaptive dictionary schemes* [2]. The task is how to create an optimal dictionary for the input text and it is NP-hard task from the size of input text [2]. There are a lot of solutions how to create the dictionary, which is quite close to the optimal solution of the problem and most of them are very close to each other in general [2]. Usually they starts from the dictionary, which consists only from the symbols of the alphabet, and to the dictionary are added all diads, then triples of symbols and so on until the predefined number of dictionary elements is reached [2].

4.3 Adaptive schemes

Most well-known dictionary algorithms are adaptive. Instead of having a completely defined dictionary when compression begins, adaptive schemes start out either with no dictionary or with a default baseline dictionary. As compression proceeds, the algorithms add new phrases to be used later as encoded tokens [1].

The basic principle behind adaptive dictionary-based compression is relatively easy to follow. They are shown in the Figure 4-2.

```
While(Not end of input stream)
{
  Word ← ReadWord(InputStream)
  If(Word ∈ Dictionary)
  {
    DictionaryIndex ← FindInDictionary(Word, Dictionary)
    Output(DictionaryIndex)
  }
  else
  {
    Output(Word)
    AddToDictionary(Word, Dictionary)
  }
}
```

Figure 4-2: Scheme of adaptive dictionary based compression.

The algorithm described below illustrates the basic components of an adaptive dictionary compression algorithm;

1. Divide the input text stream into fragments tested against the dictionary.
2. Test the input fragments against the dictionary.
3. Add new phrases to the dictionary.
4. Encode dictionary indexes and plain text so that they are distinguishable.

Almost all dictionary-based coders belongs to the same family of algorithms, based on the works of Lempel and Ziv [11,12]. The Lempel and Ziv algorithms are divided into two main parts: LZ77 [11] and LZ78 [12] algorithms.

LZ77 algorithms very often are called as *sliding window* algorithms. The main idea of this algorithm is that phrases are replaced by pointers to the previous text, and pointers indicate the place in the text, where this phrase was seen before, and the length of the text's part. By this

technique the compression method adapts very fast to the changing of the input text [2]. The principals of LZ77 algorithm are shown in the Figure 4-2.

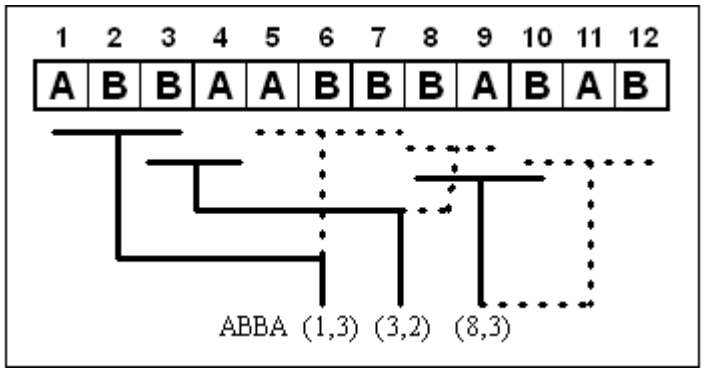


Figure 4-2: An example of LZ77 coding

LZ78 is an interesting way to adaptive dictionary compression. In this algorithm the input text is dividing into phrases, where each new phrase is the longest from previously seen plus one symbol. The phrase is coding as a pointer to the prefix plus the code of the symbol [2]. After that the new phrase is adding to the list of phrases, which can be pointed during compression. An example of LZ78 coding of the phrase "ABBAABBBABAB" is shown in the Figure 4-3.

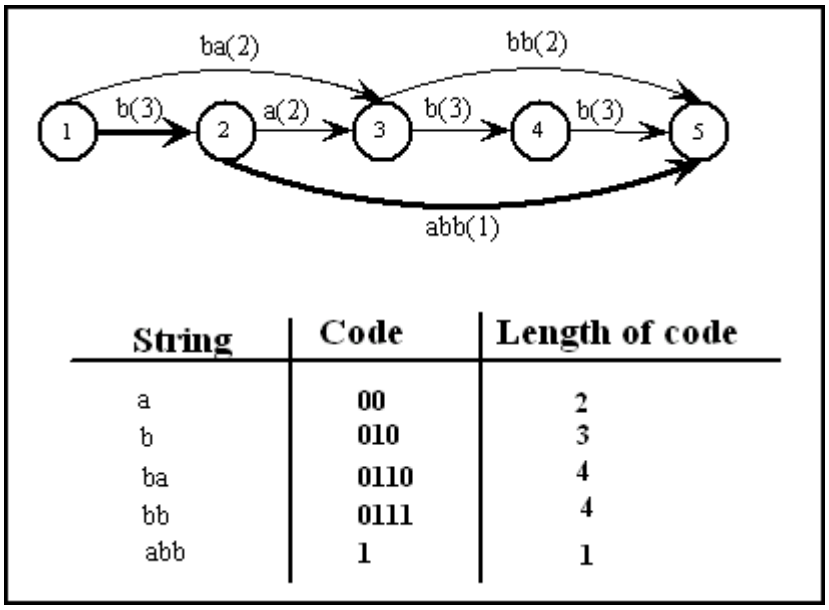
Input text	a	aa	b	ba	baa	baaa	bab
Phrase number	1	2	3	4	5	6	7
Output	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Figure 4-3: An example of LZ78 coding.

4.4 Parsing strategies

As the dictionary has been created there are several variant how to choose phrases from the input text to replace them by the dictionary's indexes. The method of dividing the input text into phrases is called *the parsing.*

- The quickest method is a *greedy* method, when the coder on each step is looking in the dictionary for a longest match. If it has no match it outputs the match, received on the previous step. The problem of greedy parsing is that it is not optimal. [2].
- The optimal parsing can be transformed to a shortest-path problem, which can be solved by existing algorithms. The transformation is performed for a string with length *n* in this way: a graph is constructed, consisting from *n*+1 nodes. For every pair of nodes *i* a directed edge is placed between then if the substring *ij* (substring starts in the point *i* string and ends in the *j*). if the corresponding string presents in the dictionary. To each edge is applied a weight, that is equal to code length of corresponding dictionary's element. An example of such process is shown in the Figure 4-4.



String	Code	Length of code
a	**00**	**2**
b	**010**	**3**
ba	**0110**	**4**
bb	**0111**	**4**
abb	**1**	**1**

Figure 4-4: Optimal parsing using a shortest path algorithm; the path weights are shown in the brackets and the shortest path is shown in boldface type.

- *LFF* (longest fragment first) method can be considered as a compromise between greedy and optimal parsing. This method is looking for the longest substring in the entrance (the coinciding can be not in the beginning of the entrance), which also presents in the dictionary. This phrase is coding and the algorithm repeats until whole entrance will be coded. For example let assume that the dictionary is formed already and it is:

  $$D = \{a, b, c, d, aa, aaaa, ab, bcb, bccb, bccba\},$$

  and all phrases are coding by four bits. If the input text is: *"aaabccbaaaa"*, the greedy parsing will output phrases: "aa, ab, c, c, baa, aa" (24 bits), LFF method will output: "aa, a, bccba, aa, a" (20 bits), and the optimal parsing gives: "aa, a, bccb, aaaa" (16 bits). In this example LFF method is somewhere between optimal and greedy methods.

Experiments showed that optimal parsing makes coding in 2-3 times slower that greedy, and increase compression on several percents [2]. LFF method increase compression, weaker than optimal but it demands much less time for coding. Usually in practice greedy parsing is the most popular, because of its speed and simplicity [2].

5. LZ77 compression method

The first compression algorithm described by Ziv and Lempel is commonly referred to as *LZ77* [11]. It is relatively simple. The dictionary consists of all the strings in a window into the previously read input stream. While new groups of symbols are being read in, the algorithm looks for matches with strings found in the previous bytes of data already read in. Any matches are encoded as pointers sent to the output stream. LZ77 and its variants make attractive compression algorithms. Maintaining the model is simple; encoding the output is simple; and programs that work very quickly can be written using LZ77. Popular programs such as PKZIP [5] and LHarc [1] use variants of the LZ77 algorithm, and they have proven very popular.

5.1 Algorithm

The main idea of LZ77 algorithm is to replace variable-length strings of input text by a triple: an offset to a phrase in the text window; the length of the phrase; and the first symbol in the look-ahead buffer that follows the phrase [1]. The amount of compression depends on how long the dictionary phrases are, how large the window into previously seen text is, and the entropy of the source text with respect to the LZ77 model. The main data structure in LZ77 is a text window, divided into two parts. The first consists of a large block of recently encoded text. The second, normally much smaller, is a look-ahead buffer. The look-ahead buffer has characters read in from the input stream but not yet encoded. The normal size of text window is several thousands characters. The size of look-ahead buffer is generally much smaller, maybe ten to one. In Figure 5-1 is shown the structure of text window.

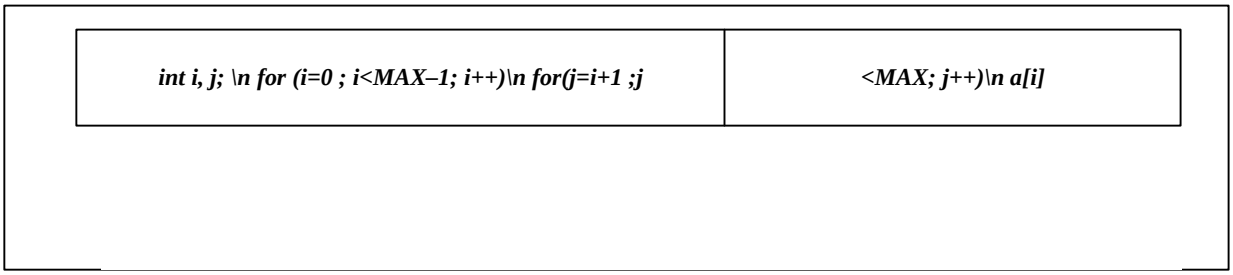


Figure 5-1: The basic structure of the text window

Here we can see that part of look-ahead buffer has a coincidence in text window — it's a string "<MAX". This string corresponds to a triple (23,4," ; "), where 23 is an index of the location of the match in the text window, 4 is the length of the match and " ; " is the first symbol after the match. LZ77 copies the match string into the text window, then shifts the text window over five characters and adds five new characters from input stream into look-ahead buffer. Then the process is repeated. The text window after shifting it by 5 symbols is shown in Figure 2.

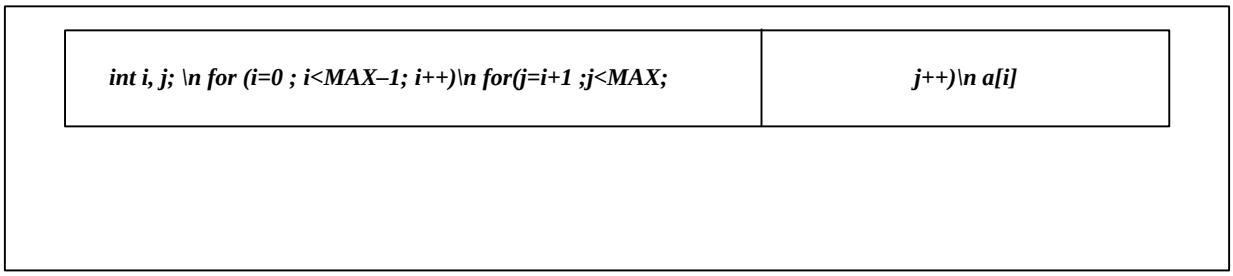


Figure 5-2: Text window after shifting of its elements

```
While(LengthOf(LookAheadBuffer)!=0)
```

```
{
  Triple ← FindTheMatch(TextWindow,LookAheadWondow)
  Distance ← GetsDistance(Triple)
  Length ← GetLength(Triple)
  FirstCharacter ← GetCharacter(Triple)
}
```

Figure 5-3. Brief pseudocode of the LZ77 algorithm.

The *Triple* in Figure 5-3 is a structure of three variables: distance, length and character.

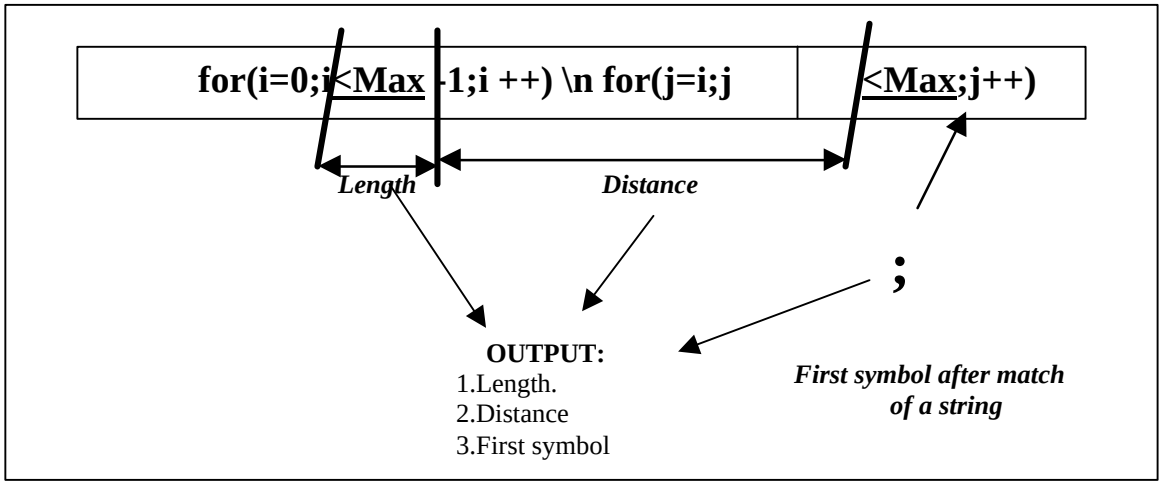


Figure 5-4: Compression process of the LZ77 algorithm.

The *Distance* can be an offset from look-ahead buffer, so it can be an offset from the beginning of the text window.

The decompression algorithm for LZ77 is even simpler, since it does not have to do any string comparisons. It reads in token, outputs the indicated phrase, outputs the following character, shifts, and repeats. It maintains the window, but it does not work with string comparisons. A decompression program that used the output of the previous program might have a view, liked is shown in the Figure 5-5. And the processes of encoding and decoding can be represented as it shown on Table5-1 and Table 5-2 consequently.

```
While (Not end of input stream)
{
  Triple ← ReadTriple (InputStream)
  Distance ← GetDistance(Triple)
  Length ← GetLength (Triple)
  FirstCharacter ← GetFirstCharacter(Triple)

  Output(TextWindow, Distance, Distance +Length)
  Output(FirstCharacter)
  ShiftTextWindow(Length+1)

  AddToTextWindow(TextWindow, Distance, Distance + Length)
}
```

Figure 5-5: Decompression process of the LZ77 algorithm.

Table 5-1: Example of LZ77 compression process for the input text “DAD DADA DADDY”

Look-ahead buffer	Text window	Match	First symbol	Output	Shift	Input symbols
“D”	“”	—	“D”	(1,0,”D”)	1	“A”
“A”	“D”	—	“A”	(1,0,”A”)	1	“D”
“D”	“DA”	“D”	—	—	0	“ ”
“D ”	“DA”	—	“ ”	(1,1,” ”)	2	“D”
“D”	“DAD ”	“D”	—	—	0	“A”
“DA”	“DAD ”	“DA”	—	—	0	“D”
“DAD”	“DAD ”	“DAD”	—	—	0	“A”
“DADA”	“DAD ”	“DAD”	—	(1,3,”A”)	3	“ ”
“ ”	“DAD DADA”	—	“ ”	(1,0,” ”)	1	“D”
“D”	“DAD DADA ”	“D”	—	—	0	“A”
“DA”	“DAD DADA ”	“DA”	—	—	0	“D”
“DAD”	“DAD DADA ”	“DAD”	—	—	0	“D”
“DADD”	“DAD DADA ”	“DAD”	“D”	(1,3,”D”)	3	“Y”
“Y”	“DAD DADA DADD”	—	“Y”	(1,0,”Y”)	1	EOF

The LZ77 algorithm encodes the string “DAD DADA DADDY” as:
(1,0,”D”), (1,0,”A”), (1,1,” ”), (1,3,”A”), (1,3,”D”), (0,0,”Y”)

Table 5-2: Example of LZ77 decompression.

Input codes	Text window	Output	Text window after shifting
(1,0,”D”)	“”	“D”	“D”
(1,0,”A”)	“D”	“A”	“DA”
(1,1,” ”)	“DA”	“D ”	“DAD ”
(1,3,”A”)	“DAD ”	“DADA”	“DAD DADA”
(1,0,” ”)	“DAD DADA”	“ ”	“DAD DADA ”
(1,3,”D”)	“DAD DADA ”	“DADD”	“DAD DADA DADD”
(1,0,”Y”)	“DAD DADA DADD”	“Y”	“DAD DADA DADDY”

5.2 Parsing of the strings

In the compression process, the algorithm needs a mechanism for finding string matches of some patterns of look-ahead buffer in text window. LZ77 uses the greedy parsing algorithm.

5.3 LZSS algorithm

LZSS [1] is a modification of LZ77 algorithm. Traditional LZ77 achieves good results of compression rapidly [1]. Even if the phrases being substituted for input text are short, they will still generally cause very effective compression to take place [1]. The problem occurs when matching phrases are not found in the dictionary. When this is the case, the compression program still has to use the same three component tokens to encode a single character. To realize the cost of this, imagine encoding a single character when using a 4096-byte window and a sixteen-byte look-ahead buffer. As 4096 is 2 raised in 12-th power so this would take twelve bits to encode a window position and another four bits to encode a phrase length. Using this system, encoding the

(0, 0, c) token would take twenty-four bits, all to encode a single eight-bit symbol. This is a very high price to pay, and there ought to be a way to improve it [1].

The main modification of this algorithm is in rejecting from the structure of triples and replacing them by new structures of data. So LZSS algorithm outputs a pair of numbers, match position index and match length, if matching have place. Otherwise it will output just a first character from look-ahead buffer. A single bit is used as a prefix that shows is it a pair offset/length or is it a simple symbol. That leads to a case when there is no use in coding string with any length as a pair index/offset, because numbers of bits that are necessary for storing in compressed file one character are less than number of bits that are required for encoding a pair index/offset. This means that we need to define a minimum match length, and if a match string has a length less that it should not be stored, but will be stored only the first symbol of look-ahead buffer. Shortly it is shown below in the Figure 5-5. Shortly the process of encoding and decoding of this modification of algorithm are described in Scheme 3 and Scheme 4 consequently.

```
While ( LengthOf(LookAheadBuffer) !=0)
{
  (Distance, Length) ← GetPair (WindowText, LookAheadBuffer)
  if (Length > MINIMUM_MATCH_LENGHT)
  {
    OutputPair(Distance, Length);
    ShiftTextWindow(Length, LookAheadBuffer)
  }
  else
  {
    Character ← GetFirstCharacter(LookAheadBuffer)
    Output(Character)
    ShiftTextWindow (1, Character)
  }
}
```

Figure 5-5: LZSS pseudocode.

Table 5-3: LZSS coding process “DAD DADA DADDY”

Look-ahead buffer	Text window	Match	Match length	Min. match length	Output	Input symbols
“D”	“”	—	0	2	“D”	“A”
“A”	“D”	—	0	2	“A”	“D”
“D”	“DA”	“D”	1	2	—	“ ”
“D ”	“DA”	“D”	1	2	“D”	“D”
“ D”	“DAD”	—	0	2	“ ”	“A”
“DA”	“DAD ”	“DA”	2	2	—	“D”
“DAD”	“DAD ”	“DAD”	3	2	—	“A”
“DADA”	“DAD ”	“DAD”	3	2	(1,3)	“ ”
“A ”	“DAD DAD”	“A”	1	2	“A”	“D”
“ D”	“DAD DADA”	“ D”	2	2	—	“A”
“ DA”	“DAD DADA”	“ DA”	3	2	—	“D”
“ DAD”	“DAD DADA”	“ DAD”	4	2	—	“D”
“ DADD”	“DAD DADA”	“ DAD”	4	2	(4,4)	“Y”
“DY”	“DAD DADA DAD”	“D”	1	2	“D”	—
“Y”	“DAD DADA DADD”	—	0	2	“Y”	—

Table 5-4: An example of LZSS decompression

Input codes	Text window	Output	Text window after shifting
“D”	“”	“D”	“D”
“A”	“D”	“A”	“DA”
“D”	“DA”	“D”	“DAD”
“ ”	“DAD”	“ ”	“DAD ”
(1,3)	“DAD ”	“DAD”	“DAD DAD”
“A”	“DAD DAD”	”A”	“DAD DADA”
(4,4)	“DAD DADA”	“ DAD”	“DAD DADA DAD”
“D”	“DAD DADA DAD”	“D”	“DAD DADA DADD”
“Y”	“DAD DADA DADD”	“Y”	“DAD DADA DADDY”

The second major change is that as in LZ77 the phrases in the text window were stored as a single contiguous block of text, with no other organization on top of it, but LZSS stores text in contiguous windows with an additional data structure that improves on the organization of the phrases.

As each phrase passes out of the look-ahead buffer and into the encoded portion of the text windows, LZSS adds the phrase to a tree structure. By sorting the phrases into a tree, the time required to find the longest matching phrase in the tree will no longer be proportional to the product of the window size and the phrase length. Instead, it will be proportional to the logarithm of the window size multiplied by the phrase length.

The savings created by using the tree not only makes the compression side of the algorithm much more efficient, it also encourages experimentation with longer window sizes. Doubling the size of the text window now might only cause a small increase in the compression time, whereas before it would have doubled it [1].

5.4 Deflate algorithm

Deflate algorithm is a modification of LZ77 algorithm [4]. It’s realized in such well-known compressor as GZIP [4] and in well-known image representation format as PNG [6]. This algorithm is interesting from the point of view of implementation of LZ77 algorithm.

It would be useless to describe all details (they are described much better in specification of algorithm [2]) of algorithm because there are no principal new ideas about encoding of data, but some good ideas about realization of LZ77 algorithm.

Compressed data consists of blocks, where blocks could represent three different types of data:

1. Stored data. This is data that were not compressed.
2. For encoding data was used fixed Huffman coding, defined by Deflate algorithm specification.
3. For encoding was used dynamic Huffman codes.

Remark: all numbers below were taken form “deflate” specification.
Encoded data blocks in the "deflate" format consist of sequences of symbols drawn from three conceptually distinct alphabets: either literal bytes, from the alphabet of byte values (0..255), or pairs, where the length is drawn from (3..258) [4] (255 — size of look-ahead buffer and 3 is

minimum match length) and the distance is drawn from (1..32,768) (here 32,768 — size of text window) [2]. In fact, the literal and length alphabets are merged into a single alphabet (0..285), where values 0..255 represent literal bytes, the value 256 indicates end-of-block, and values 257..285 represent length codes (possibly in conjunction with extra bits following the symbol code) as in Table 5-5 and distances in LZ77 algorithm are coded by Table 5-6, where extra bits defines a number in the intervals of lengths or distances:

Table 5-5: Length codes

Code	Extra code bits	Length(s)		Code	Extra code bits	Length(s)
257	0	3		272	2	31-34
258	0	4		273	3	35-42
259	0	5		274	3	43-50
260	0	6		275	3	51-58
261	0	8		276	3	59-66
262	0	9		278	4	83-98
263	0	9		279	4	99-114
264	0	10		280	4	115-130
265	1	11,12		281	5	131-162
266	1	13,14		282	5	163-194
267	1	15,16		283	5	195-226
268	1	17,18		284	5	227-257
269	2	19-22		285	0	258
270	2	23-26				
271	2	27-30				

Table 5-6: Distance codes

Code	Extra bits	Distance(s)		Code	Extra bits	Distance(s)		Code	Extra bits	Distance(s)
0	0	1		10	4	33-48		20	9	1025-1536
1	0	2		11	4	49-64		21	9	1537-2048
2	0	3		12	5	65-96		22	10	2049-3072
3	0	4		13	5	97-128		23	10	3073-4096
4	1	5,6		14	6	129-192		24	11	4097-6144
5	1	7,8		15	6	193-256		25	11	6145-8192
6	2	9-12		16	7	257-384		26	12	8193-12288
7	2	13-16		17	7	385-512		27	12	12289-16384
8	3	17-24		18	8	513-768		28	13	16385-24576
9	3	23-32		19	8	768-1024		29	13	24577-32768

In case of dynamic Huffman coding in output file are going codes, which are coded by Huffman code. In case of fixed Huffman codes for literal, length and distance codes used codes from Table 5-7. After them are going out extra bits (if necessary), which carry information about length or distance.

Table 5-7: Fixed Huffman codes

Literal codes	**Bits**
0-143	8
144-255	9
256-279	7
280-287	8
Distance codes	**Bits**
0-31	5

Deflate algorithm during has one important modification in substring searching process: after it finds a match, it doesn't stop, but tries to find a longer match, starting to search from next byte of look-ahead buffer. If the next parsing is successful, then the previously matched string is truncated to 1 byte (a simple literal), this byte is coded, and we define a new match as match string and process repeats. Otherwise we output pair length/distance.

5.5. PNG image representation format

PNG (Portable Network Graphics) is an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF file format. It is interesting for us from the point of view of its implementation of its compression algorithm. The algorithm, used in the PNG, is standard LZ77 algorithm with deflate modification. Deflate compression also used in zip, gzip, pkzip and related programs. PNG is designed to work well in online viewing applications, such as the World Wide Web, so it is fully streamable with a progressive display option [5]. As far as PNG was developed to replace GIF so it has all features that GIF has, but PNG also has some additional features. PNG is designed to be simple and portable, that is why developers are able to implement PNG easily with the help of standard libraries.

6. LZ78 algorithm

LZ78 is similar to LZ77 in some ways. LZ77 outputs a series of tokens. Each token has three components: a phrase location, the phrase length, and a character that follows the phrase. LZ78 also outputs a series of tokens with essentially the same meanings. Each LZ78 token consists of a code that selects a given phrase and a single character that follows the phrase. Unlike LZ77, the phrase length is not passed since the decoder knows it. Unlike LZ77, LZ78 does not have a ready-made window full of text to use as a dictionary. It creates a new phrase each time a token is output, and it adds that phrase to the dictionary. After the phrase is added, it will be available to the encoder at any time in the future, not just for the next few thousand characters.

6.1 Compression algorithm

When using the LZ78 algorithm, both encoder and the decoder start off with a nearly empty dictionary. By definition, the dictionary has a single encoded string—the null string. As each character is read in, it is added to the current string. As long as the current string matches some phrase in the dictionary, this process continues. But at this point, LZ78 takes an additional step. The new phrase, consisting of the dictionary match and the new character; is added to the dictionary. The next time that phrase appears, it can be used to build an even longer phrase. The pseudocode of the compression algorithm is shown Figure 6-1, and its block-scheme is shown in the Figure 6-2.

```
String ← ReadCharacter(InputStream)
While (Not end of InputStream)
{
  Character ← ReadCharacter(InputStream)
  if(String +Character ∉ CodeTable)
  {
    String ← String + Character
  }
  else
  {
    Output( Code (String))
    AddIntoTheCodeTable(“String + Character”)
    String ← Character
  }
}
Output(Code(String)).
```

Figure 6-1: A pseudocode for LZ78 compression**.**

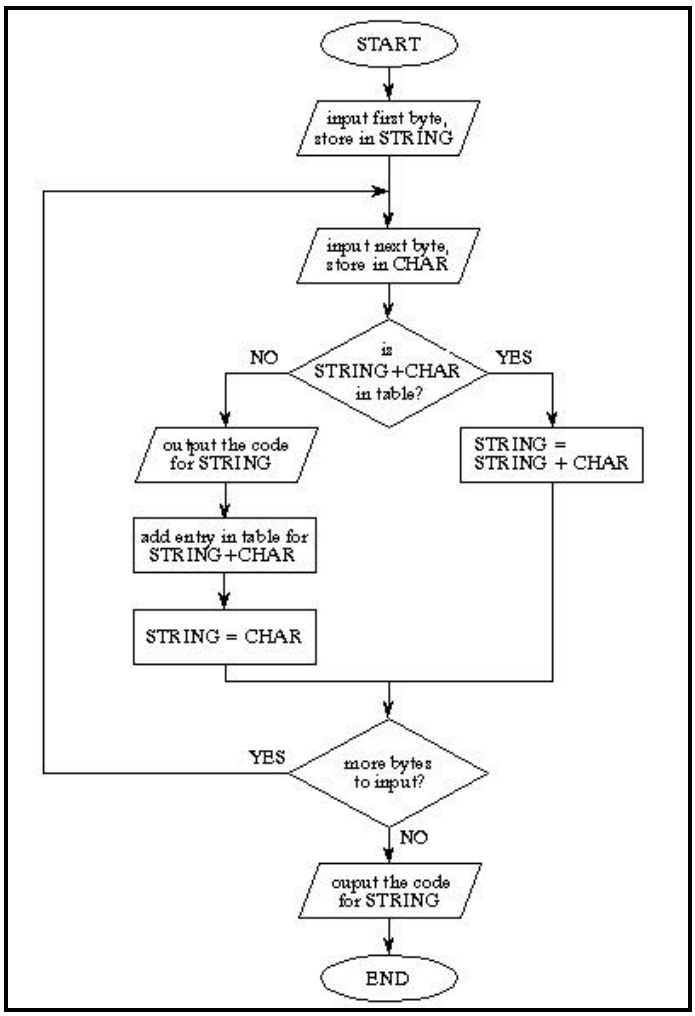


Figure 6-2: Block-scheme for LZ78 compression process.

Table 6-1: LZ78 resulting dictionary for input string input text is "*DAD DADA DADDY DADO*"

Output Phrase	Output Character	Encoded String	Code of encoded String
0	'D'	"D"	1
0	'A'	"A"	2
1	' '	"D "	3
1	'A'	"DA"	4
4	' '	"DA "	5
4	'D'	"DAD"	6
1	'Y'	"DY"	7
0	' '	" "	8
6	'O'	"DADO	9

The dictionary, created during the process of compression is shown below, in Table 6-1. The first two characters to come through the encoder, 'D' and 'A,' have not been seen before. Each will have to be encoded as a phrase, 0+ character pair. "D" is added to the dictionary as phrase 1, and "A" is added as phrase 2. When the third character, 'D,' is read in, it matches an existing dictionary phrase. The ' ' character, the next character read in, creates a new phrase with no match in the dictionary. LZ78 will output code 1 for the previous match (the D string), then the " " character. The overall tree structure of LZ78 dictionary is shown in Figure 6-3.

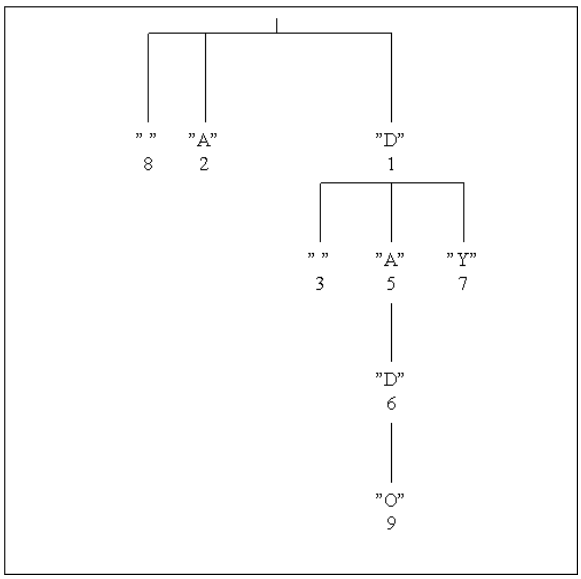


Figure 6-3: LZ78 resulting tree.

6.2 LZW algorithm

As LZ77 has LZSS modification, so LZ78 has also it's own modification known as LZW [13]. LZSS improved on LZ77 compression by eliminating the requirement that each token output a phrase and a character. LZW makes the same improvement on LZ78. In fact, under LZW, the compressor never outputs single characters, only phrases. To do this, the major change in LZW is to preload the phrase dictionary with single-symbol phrases equal to the number of symbols in the alphabet. Thus, there is no symbol that cannot be immediately encoded even if it has not already appeared in the input stream.

An example of encoding process is shown below in Table 6-2. In this example an alphabet is "A", "D", "Y" and their indexes are consequently 1, 2, 3.

Table 6-2: An example of LZW compression process.

Input: "DAD DADA DADDY"		
Characters Input	**Code Output**	**New code value and associated string**
"D"	—	—
"DA"	2	4 = "DA"
"AD"	1	5 = "AD"
"D "	2	6 = "D"
" D"	2	7 = " D"
"DA"	—	—
"DAD"	4	8 = "DAD"
"DA"	—	—
"DA "	4	9 = "DAD "
" D"	—	—
" DA"	7	" DA"
"AD"	—	—
"ADD"	5	10 = "ADD"
"DY"	2	11 = "DY"
EOF	3	—

6.3 Decompression

One reason for the efficiency of the LZW algorithm is that it does not need to pass the dictionary to the decompressor. As for LZ78 and LZW the decompression algorithm is the same: difference only in the initial dictionaries. The table can be built exactly as it was during compression, using the input stream as data. This is possible because the compression algorithm always outputs the phrase and character components of a code before it uses it in the output stream, so the compressed data is not burdened with carrying a large dictionary. This algorithm in pseudocode is shown below in the Figure 6-4:

```
OldCode ← Read(InputStream)
OldString ← GetDictionaryValue(OldCode)
Output(OldString)
While (Not end of InputStream)
{
  NewCode ← Read(InputStream)
  NewString ← GetDictionaryValue(NewCode)
  Output(NewString)
  OldString←FirstCharacter(NewString )
  AddToDictionary (OldString) ;
  OldString ← NewString
}
```

Figure 6-4: The decompression LZ78 algorithm**.**

Unfortunately, the decompression algorithm shown is just a little too simple. A single exception in the LZW compression algorithm causes some trouble in decompression. Each time the compressor adds a new string to the phrase table, it does so before the entire phrase has actually been output to the file. If for some reason the compressor used that phrase as its next code, the expansion code would have a problem. It would be expected to decode a string that was not yet in its table. It's so-called special case:

CHARACTER+STRING+CHARACTER+STRING+CHARACTER

It's the case when LZW compressor outputs a code before LZW decompressor can decode it (see Table 6-3). Here alphabet is A – 1, B – 2. The process of decompression for this case is shown in Table 6-4. The pseudocode of working decompression in LZW coding is placed in the Figure 6-5. In Figure 6-6 is placed a block-scheme of decompression process in LZW algorithm.

Table 6-3: Encoding for input string "*AAAB*"

Character Input	New code value and associated string	Code Output
A	—	—
A	3 — AA	1
A	—	—
B	4 — AAB	3

Table 6-4: Decoding process for previously coded string.

Code Input	New code value and associated string	Output
1	3 = A + …	A
3	3 = A + 3	?

```
OldCode ← Read(InputStream)
OldString ← GetDictionaryValue(OldCode)
Output(OldString)
While (Not end of InputStream)
{
  NewCode ← Read(InputStream)
  if ( NewCode ∉ CodeTable )
  {
    String ← GetDictionaryValue (OldCode)
    String ← String + Character
  }
  else
  {
    String ← GetDictionaryValue (NewCode)
  }
  Output (String)
  Character ← GetFirstCharacter(String)
  AddEntryToDictionary(OldCode + Character)
  OldCode ← NewCode
}
```

Figure 6-5: A pseudocode of LZ78 decompression algorithm.

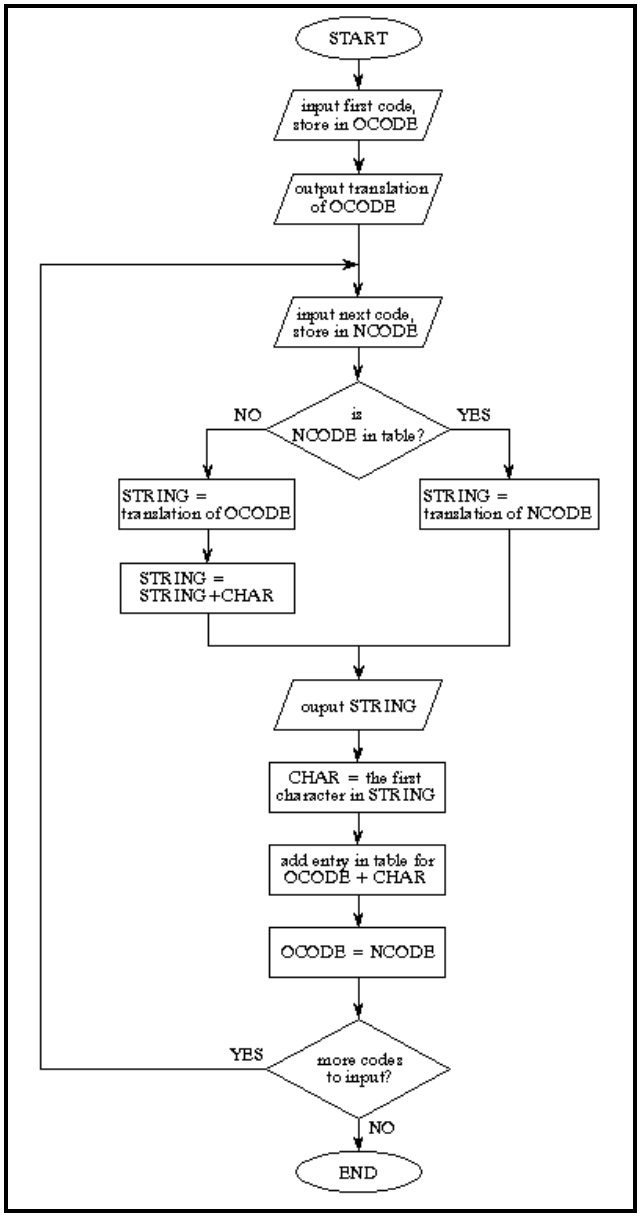


Figure 6-6: A bock-scheme that describes the LZ78 decompression process.

6.4 Coding of the pointers

One negative side effect of LZ78, which is not found in LZ77, is that the decoder has to keep up the tree in exactly the same fashion as the encoder, or a disastrous mismatch will occur. In case of LZ77 a dictionary index was just a pointer or a index to the previous position in the data stream. But in the case of LZ78 the index is the number of a node in the dictionary tree. And therefore decoder has to maintain this dictionary tree as well.

Another issue ignored so far is that of the dictionary is filling up. Regardless of how big the dictionary space is, it is going to fill up sooner or later. If we are using a twelve-bit code, the dictionary will fill up after it has 4096 (because $2^{12} = 4096$) phrases defined in it.

There are several alternative choices regarding a full dictionary. Probably the safest default choice is to stop adding of new phrases into the dictionary after it is full. But this is not the best choice as statistical model can completely change and our dictionary would be unacceptable for given file. In other hand the *Shrinking method* [5] is a LZW compression algorithm with partial clearing of the dictionary. Shrinking differs from conventional LZW implementations in several respects:

1. The code size is controlled by the compressor, and is automatically increased when codes larger than the current code size are created (but not necessarily used). When the decompressor encounters the special code sequence (for example 256 followed by 1) it should increase the code bit size and read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the special sequence (256,1) is encountered. [5]
2. When the table becomes full, total clearing is not performed. Rather, when the compressor emits another one special code sequence (for example 256,2), the decompressor should clear all leaf nodes from the LZ tree, and continue to use the current code size. The nodes that are cleared from the LZ tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence (256,2) at any time. [5]

Briefly this method consists in cutting off all leaves from LZ78 or LZW dictionary tree (see Figure 6-5). This procedure proceeds because as far as the dictionary does not create new elements so the compression stops to adapt to the input stream of symbols. To prevent this situation next action performed: all nodes that have not children nodes are deleted. This is legal as in LZW algorithm code can be used for encoding only in the moment of creating its children nodes.

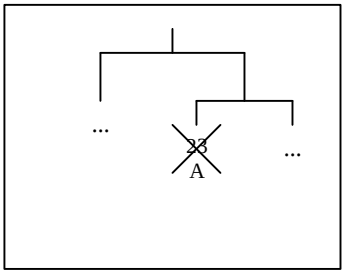


Figure 6-5: LZ78 tree after cleaning: node 23 had no children, that why it's deleted, index 23 is free and can be reused.

Shrinking method can be modified in the simple way. We just do not need to delete all leave nodes at one moment. The process of cleaning can be divided into several iterations, and during each of iteration it can delete only one leaf with lowest index and reuse only one index of dictionary at each iteration. Another leaves have a “hope” that they will become “not leaf”, that is why new elements of the dictionary could have bigger length and compression rating could also be better. Simple the difference between two methods is shown in Figure 6-6, 6-7, 6-8.

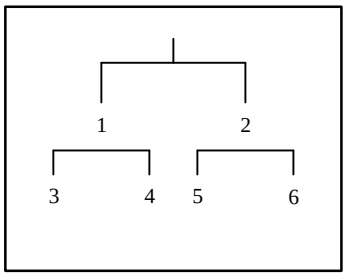


Figure 6–6: Dictionary tree before cleaning.

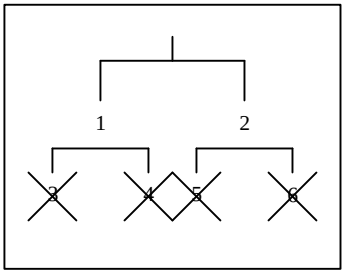


Figure 6-7: Cleaned dictionary tree in Shrinking method.

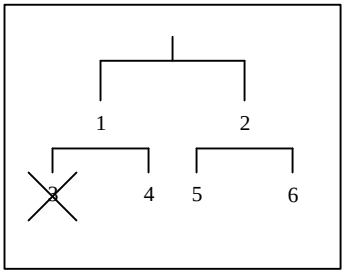


Figure 6-8: Cleaned dictionary tree in modified Shrinking method.

7. Application to map image compression

7.1 Personal navigation

Digital maps provide visual view on a given geographic location that can be used for application dealing with spatial data in *personal navigation*. We can select user-specific views of the map server for different applications. The main goal is to have the maps available in real-time and independent of the location of the user, and without excessive computing resources. Typical navigation devices have limited memory resources and very narrow wireless communication channel [8].

7.2 Map images

For map representation usually used three types of images: binary images, greyscale and colour. Some words about colour representation in a image: a colour perceived by the human eye can be defined by a linear combination of the three primary colours: red, green and blue. These three colours form the basis for the RGB-colour space. Hence, each perceivable colour can be defined by three values, which represent the red, green and blue component consequently.

Binary images are images whose pixels have only two possible colours. They are normally displayed as black and white. Numerically, the two values are used: 0 for black, and either 1 or 255 for white. Binary images are often produced by tresholding of a greyscale or colour image, in order to separate an object in the image from the background. The colour of the object (usually white) is referred to as the *foreground colour*. The rest (usually black) is referred to as the *background colour*.

A greyscale (or greylevel) image is simply one in which the only colours are shades of grey. The reason for differentiating such images from any other sort of colour image is that less information needs to be provided for each pixel. In fact, grey colour is one in which the red, green and blue components all have equal intensity in RGB-space, and so it is only necessary to specify a single intensity value for each pixel, as opposed to the three intensities needed to specify each pixel in a colour image.

In a colour image a pixel consists of three values: red, green, blue. For each of the values is used 8 bits, so each pixel is represented by 24 bits and total number of colours in the 16777216 (2^{24}).

7.3 Compression of maps

The storage size of a map is huge. For example, electronic library of Finnish road maps of the resolution 1:250 000 takes entire CD (over 600 Mb) in uncompressed form [7CMI]. In comparison, the portable viewing device, such as pocket computer, have about 32 Mb of storage space, which can be expanded by about 96 Mb through using compact flash memory cards [9]. The storage requirements of the maps can be therefore be a bottleneck, especially in the case of portable devices, in which the maps share the limited memory with the operating system, application and other data.

Because of limited resources, the end-user cannot store the large map database with specific software for the database management. A much simpler but still useful approach is to provide the user with digital compressed images. In this way, the user will get almost all the same benefits as he would get from the access to traditional geographic information systems with map databases. In additional to that, the user would not need to have the huge hardware and software resources. For example, an uncompressed black-and-white image of 5000x5000 pixels takes about 3

megabytes in uncompressed form. Using a state of art compression technology [9], this image can be compressed usually by a factor 10:1, which corresponds to the file size of 300 Kb. A drawback of the existing compression techniques is that entire image must be decompressed in memory before the image can be presented to the user [9]. This can be a problem, as device may not have sufficient resources for real-time decompression of the image.

The static models of compression have the smallest demands to resources. The semi-adaptive compression models take the second placed in the resource demands, as the all information, which was created during compression is transferred into the compressed file. The adaptive models are the most demanding to the resources, as they need to reconstruct the model during decompression.

7.4 Block decomposition for direct access

The input map images are divided into *bxb* non-overlapping rectangular before the compression, and each block is compressed separately from other as proposed in [9] (see Figure 7-1). The compression blocks are stored in the same file, and an index table is stored in the header of the file. When the compressed map image is accessed, a block index table indicating the location of the block in the compressed file, can be constructed. This provides direct access to the compressed image file, and therefore, enables efficient and independent decompression of particular image fragment. The block size is a trade-off between compression efficiency and decoding delay. If very small block size is used the desired part of the image can be reconstructed more accurately and faster. The compression, however, would be less efficient because of a less accurate probability model. The index table itself requires space and the overhead is relative to the number of blocks.

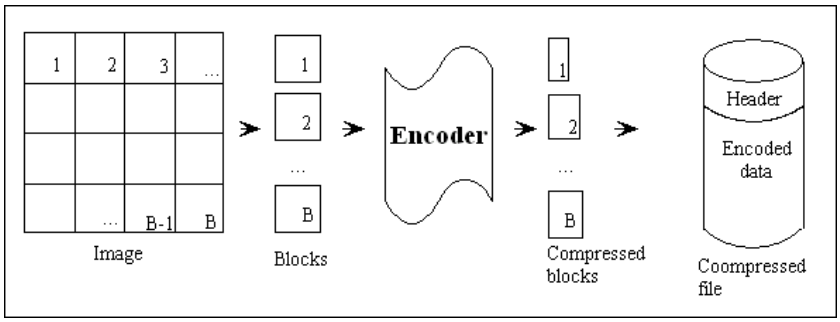


Figure 7-1: Diagram of the block decomposition

8. Semi-adaptive LZW

In previous chapter was said that the block-segmentation is a compromise between compression ratio and decompression delay. It means that when the decompression delay becomes smaller, the compression ratio also decreases. Is it possible to create an algorithm, where the dependency between them is not so direct? Such algorithm requires a modelling that takes into account as the global modelling for the whole image, as the local modelling for each of the blocks. Above were described two main ideas of modelling: statistical modelling and dictionary schemes. Both modelling can operate as globally, as locally. In this chapter we consider a new idea denoted as semi-adaptive LZW coding will be considered.

8.1 Ideology of semi-adaptive LZW

The LZW coding is a good compression algorithm, which can adapt itself to the changes in the input stream [1]. The main achievement of this algorithm is that it is from the family of adaptive coders that let it not to store the dictionary into the file, but create it during decompression process. It is very important, because the dictionary can easily reach very huge size. On the other hand, the abilities in compression of LZW algorithm will decrease if we use it for compression maps with small number of colours (binary images for example), or texts with small number of different used characters. This is because the structure of LZW tree, where the number of children of each root cannot be more than the number of different elements of the input stream. That is why during compression of the file with small number of different elements, binary images for example, size of the dictionary increases very fast and compression must be continued with that dictionary which we have, or forget about the dictionary and start to create a new dictionary. It is clear that it is not the best way for using the dictionary.

Here we present a modification of LZW, which is denoted as *LZWsem* (LZW semi-adaptive). In the standard LZW the algorithm process without pre processing phase. The LZWsem has two phases of compression: in the first phase it starts to create the dictionary, and the actual output of the indexes is started in the second phase. *LZWsem* uses in the preprocessing phase the same algorithm of the dictionary creating process, like in LZW. During the second stage it uses simple greedy algorithm. But as the LZW is semi-adaptive the output will consist of the compressed information plus the dictionary.

Let us consider an example. So if the initial alphabet is 0 – A, 1 – B, 2 – C, 3 – D, and the input is**: ”***AAABBACDBBAACDA***”,** then the result of the adaptive coding is shown in Table 8-1, and the final dictionary is shown in Table 8-2. Let us overview the coding of the input by this dictionary (see Table 8-3).

Table 8-1: Process of compression in adaptive variant.

Input character	Output code	New dictionary element	Current code bits
A	—	—	
A	0	4. AA	3
A	—	—	3
B	4	5.AAB	3
B	1	6.BB	3
A	1	7.BA	3
C	0	8.AC	4
D	2	9.CD	4

B	3	10.DB	4
B	—	—	4
A	6	11.BBA	4
C	—	—	4
D	8	12.ACD	4
A	3	13.DA	4
—	0	—	4

Table 8-2: Full dictionary created by LZW algorithm.

Code	Value
0	A
1	B
2	C
3	D
4	AA
5	AAB
6	BB
7	BA
8	AC
9	CD
10	DB
11	BBA
12	ACD
13	DA

Table 8-3: Semi-adaptive coding process. Input text: ”AAABBACDBBAACDA**”**

Input character	Current string	Current code	Output code
A	A	0	—
A	AA	4	—
A	AAA	—	4
B	AB	—	0
B	BB	6	—
A	BBA	11	—
C	BBAC	—	11
D	CD	9	—
B	CDB	—	9
B	BB	6	—
A	BBA	11	—
A	BBAA	—	11
C	AC	8	—
D	ACD	12	—
A	ACDA	—	12
“”	A	0	0

The shrinking is used in both compression methods: in adaptive and in semi-adaptive, and the comparing will be done on the results of the compression. As it was said before, for first 2 elements of dictionary we need only 1 bit (as they are 0 and 1) to store in the file, for next 2 elements are necessary 2 bits, next 4 elements requires 3 bits, and so on.

Table 8-4: Final dictionary for semi-adaptive coding.

Old code	New code
0	0
1	1
2	2
3	3
4	4
11	5
9	6
12	7

Unlike adaptive LZW method in semi-adaptive modification it is necessary to increase code bit length only before outputting a code with larger bit length (the coder knows about it every time when it happens, but in this case the program needs some reserved symbols to encode the increasing of the code bit length).

Table 8-5: Comparing of two compression methods.

Adaptive LZW		**Semi-adaptive LZW**	
Code	**Code bit length**	**Code (New code)**	**Code bit length**
0	3	4(4)	3
4	3	0(0)	3
1	3	11(5)	3
1	3	9(6)	3
0	4	11(5)	3
2	4	12(7)	3
3	4	0(0)	3
6	4	—	—
8	4	—	—
3	4	—	—
0	4	—	—
Size of compressed sequence			
Adaptive LZW		**Semi-adaptive LZW**	
40 bits		21 bits	

Most part of such optimistic effect, of course, will be neglected for big sequences, but the main idea will stay: semi-adaptive LZW algorithm uses less number of indexes for encoding and that is why it has a right for existing.

The implementation of the semi-adaptive method can be divided into several parts. The first part is creation of the initial LZW dictionary. It means that at the first time the algorithm must process the simple adaptive LZW coding process, and create dictionary so called as *initial dictionary*. The next step is pruning of the initial dictionary. It includes gathering of the statistics for each dictionary element. After this, for each initial dictionary element the decision must be made: is it useful to keep this element in the final version of the dictionary, or it would be more useful to get rid of this element. After this, the dictionary must be stored in the compressed file. Also it would be very useful to use some kind of compression to for storing the dictionary. At the final step the algorithm must code the input file by using the final version of the dictionary. And if it is so the information about the dictionary compression also must be kept into the compressed file.

The algorithm, that will be described later, will have the following steps:

1. Creation of the initial LZW dictionary.
2. Pruning of the initial dictionary and calculating statistics of all elements from the final dictionary.
3. Storing the final version of the dictionary in the compressed file.
4. Encoding the input file by the final version of the dictionary.

8.2 The creation of the initial dictionary

The initial dictionary creation process is predefined before the entire workflow of the semi-adaptive LZW algorithm. There three options to choose, and the algorithm can operate by three different variants of initial dictionary creating process.

1. To find all used diads, triads and so on.
2. To process initial dictionary creating phase separately from block to block.
3. To process the first phase for the whole image.

The first one is to find over the whole input image all used diads (a sequence of symbols with length two, triads (a sequence with length three) and so on. During this process the encoder goes several times trough the input image. This algorithm is very close to usual LZW algorithm, but in its case the number of the iteration limits the maximum length of the codeword during iteration.

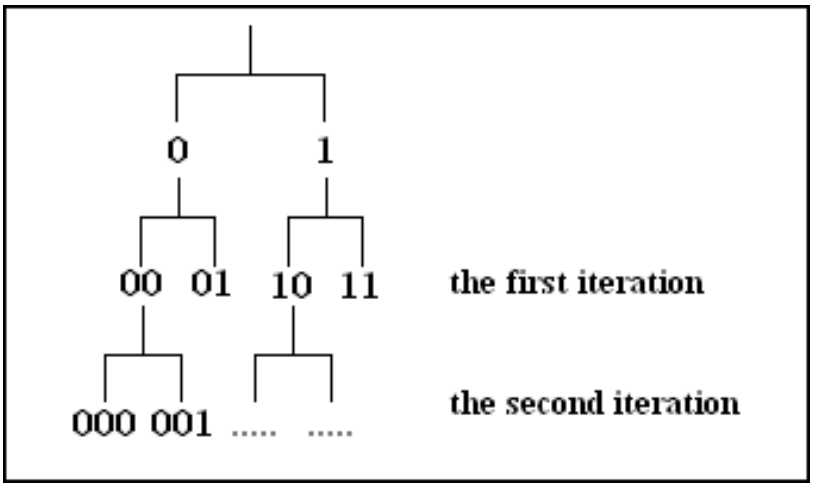


Figure 8.1: The LZW tree in the first case type of binary image coding.

In the Figure 8-1 is shown the LZW tree in the case of first type of encoding. To reduce the number of iterations the iteration stops each time when all codewords with fixed length are found. The maximum number of codewords in the iteration n is m^n, where m is the amount of colours in the input image.

The second way is to predefine the number of LZW tree nodes per blocks. For example if compression process operates by 4096 nodes in the LZW tree, and the input image is divided into 4 blocks, then for each block only 1024 nodes ($1024 \cdot 4 = 4096$) must be used. When 1024 nodes are processed, the initial dictionary creating process continues in the next block of the input image. If during the initial dictionary creating process over the current block less than predefined number of the nodes of the initial dictionary's LZW tree were created, then the process repeats from the beginning of the block. The initial dictionary keeps its structure from block to block and all nodes that were created on previous steps are used for creating of new nodes (see example of such processing in Table 8-6).

Table 8-6:An example of the initially dictionary-creating process with two processing over input text: "**AABBCCAA**"

First phase of processing		
Input Character	Encoded String	Code of encoded String
A	A	“”
A	AA	4
B	AB	5
C	BC	6
C	CC	7
A	CA	8
A	AA	“”
Second phase of processing		
A	AAA	9
A	AA	“”
B	AAB	10
B	BB	11
C	BC	“”
C	BCC	12
A	CA	“”
A	CAA	13

The third way is to forget about the blocking structure of the input image and create the initial dictionary in the traditional LZW way. In this case the program starts to create LZW dictionary from the beginning of the input image and continues process until the end of the image. When the number of initial dictionary elements reaches the predefined maximum number of elements the process stops. Otherwise, if during initial dictionary creating process the processing over whole image does not reach this number, the process begins from the start of the input image again. The initial dictionary elements, which were created during previous passing over the input image, take part in the creating of new elements in new passing over the input image.

8.3 Pruning of the initial dictionary

The initial dictionary is the basement for the *final dictionary*. Elements from the final dictionary are used for encoding input image in the final stage. Transformation of the initial dictionary into the final dictionary consists of deleting the most of elements from the initial dictionary. The process consists from two stages:

1. Cover creating.
2. Pruning of the cover.

At the first stage all used elements are found. The algorithm of this process is described in the Figure 8-2. Briefly, the first stage can be described in the following way: the algorithm starts to create a *cover* of the input image. Elements of the cover are taken from the initial dictionary. And all elements from the initial dictionary are divided now into two groups: from the covering (used ones) and not from the covering (unused ones). The unused elements are not going to present in final dictionary.

To describe the second stage of pruning it is necessary to repeat some info about the shrinking algorithm. As it was said before in previous chapter, the indexes of codes have not fixed code bit length in the shrinking algorithm, but the code bit length is increasing during increasing of the indexes. During the first stage of pruning the indexes, which were set to the elements of the final

dictionary, they have the same order as the values of the elements in the input image. For example, the first codeword, which is appeared during first stage, will have index 0, the second one index 1, and so on. This means that for first two indexes it is possible to use 1 bit, for the next 2 indexes (which are 2 and 3) is possible to use 2 bits, and so on. So during the compression so-called *current code bit length*, which defines the number of bits that are using in current time to output a codeword, is changing. If there is a situation when the current code bit length is more than sufficient to encode the current code (for example: current code is 1 and current code bit length is 9, but for encoding 1 is necessary only 1 bit), the current code is encoded anyway by the current code bit length bits. In shrinking algorithm current code bit length increases only when the number of created indexes is more than $2^{CurrentCodeBits}$, where variable *CurrentCodeBits* equals to the current code bit length.

```
Read( FirstSymbol )
CurrentString ← FirstSymbol
CurrentStringIndex  ← 1
While( Not end of the input stream )
{
Read(Symbol)
if(CurrentString + Symbol ∉ InitialDictionary)
      {
        Index(CurrentString) ← CurrentStringIndex + +;
        CurrentString ← Symbol
      }

else
      {
      CurrentString ← CurrentString + Symbol
      }
}
```

Figure 8-2: The pseudocode for the first stage of the pruning of the dictionary

During the second stage of pruning the role of indexes is depends on the variable *CurrentStringIndex.* According to the CurrentStringIndex the program calculates each time the CurrentCodeBits variable. For each dictionary two values are calculated:

1. *ElementCount*: the number of appearance of the code from initial dictionary in the covering of the input image.
2. *ElementBitCount*: the sum of code bit lengths in all moments when the code appeared during the first stage.

The node from the cover (of course it also belongs to initial dictionary) is stayed in the final dictionary if it is satisfied to the formula:

$$DepthOfNode \cdot \log_2(ColoursNumber) \cdot (ElementCount - 1) > ElementBitCount \qquad (7.3)$$

where *DepthOfNod*e is the depth of the node in the LZW tree of the initial dictionary, *ColoursNumber* is the number of colours in colour palette. As the value of *DepthOfNod*e is same as the length of the word, which that is represented by this node, the formula (7.3) can be explained in such way: if an index is used to encode a sequence of symbols (in this case symbols are pixel's indexes), the sum of bits, which are used for it, must be less than the number of bits, which are required to output the sequence without encoding. There is used $ElementCount - 1$ in the left side of the formula, because in semi-adaptive LZW algorithm for each new element of

the final dictionary some bit space must be used for storing the dictionary into the compressed file. The results of pruning of the initial dictionary are placed in Table 8-7.

Table 8-7: The results of pruning of the initial dictionary.

Image name	**Number of nodes**		
	before pruning	**After 1-st stage**	**After 2-nd stage**
v11.pgm	512	394	201
vantaa.pgm	512	483	426
suomi2-q.pgm	512	450	353
zzz.pbm	512	327	325

8.4 The final dictionary structure

The dictionary in traditional LZW algorithm is not stored and it is usual for any adaptive dictionary-based algorithm. But in the case of semi-adaptive LZW algorithm it is necessary to store whole dictionary in the compressed file. There are two methods to store the final dictionary elements into the compressed file:

1. To keep the tree structure of the LZW tree, the final dictionary is stored in the compressed file like a tree (pointer to the parent node + symbol).
2. To get rid of the tree structure and store the final dictionary as a simple codebook.

In the first case, because it is necessary to keep the whole tree structure, the element of the dictionary is consisting from: $parent + symbol$. Size of parent is calculating from the predefined number of LZW tree nodes, for example if LZW tree consists from 512 nodes, then it is necessary to use 9 bits to encode one index of the nodes $(\log_2 512 = 9)$. The code bit length of the symbol depends from the number of colours. So if the image has 8 colours, it is necessary to use at least 3 bits to encode the index of colour $(\log_2 8 = 3)$. The final formula for the size of the transformed information is about:

$$(\log_2 ColoursNumber + \log_2 NodesNumber) \cdot NodesNumber \text{ bits}, \tag{8.1}$$

where *ColoursNumber* is the number of colours in the colour palette of the input image and *NodesNumbe*r is the predefined number of nodes in the initial dictionary LZW tree.

In the second variant the formula for the size of transmitted information is about:

$$\sum_{i=1}^{UseNodesdNumber} DepthOfNode_i \cdot \log_2 ColoursNumber \text{ bits}, \tag{8.2}$$

where *UsedNodesNumbe*r is the number of nodes from the initial dictionary that were used in the cover of the input image, *DepthOfNode* is the depth of the node in the LZW tree (or length of the phrase, which it represents). For example, let us consider the LZW tree received after the example, which is described in the first paragraph of the current chapter before pruning and after it (see Figure 8-3 and Figure 8-4 consequently).

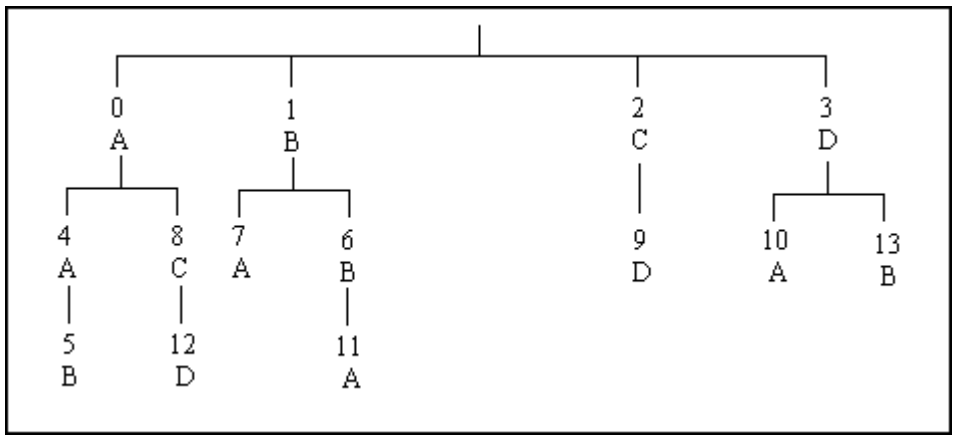


Figure 8-3: LZW tree for input string "*AAABBACDBBAACDA*"**.**

Indexes	Codewords
0	A
2	C
3	D
4	ACD
5	BBA
6	CD

Figure 8-4: The result codebook from the previous example, which represents the LZW tree after first stage of pruning.

So, in the first case the amount of space, necessary to store the tree structure is (here: *NodesNumber* is 13, *ColoursNumber* is 4):

$$(2+4)\cdot 13 = 78 \text{ bits.}$$

And in the second case the amount of space is (*NodesNumber* is 7, *ColoursNumber* is 4):

$$(1+1+1+1+2+3+3)\cdot 2 = 28 \text{ bits}$$

All conclusions below are based on the series of experiments. The results of these experiments placed in Table 8-6 (The testing images can be seen in Appendix, part A).

Table 8-6: Result of experiments in comparing two variant of storing the dictionary

Image	**Nodes Number**	**Colours Number**	**Used Nodes Number**	**Variant1 Size**	**Variant2 Size**
v11.pgm	512	4	201	5632	3198
vantaa.pgm	512	4	426	5632	5570
suomi2-q.pgm	512	8	353	6144	5928
zzz.pbm	512	2	325	5120	3505

It can be seen from the table that on the testing images the second variant is better than the first one. By the way, the results could be different for the images with big colour palette.

With increasing the number of LZW tree nodes the advantage of the second variant is getting clearer, moreover there is a tendency that the difference between sizes of transmitted information

increases in the favour of the second variant. The tendency is shown in Table 8-7. This shown series was passed over the image v11.pgm, which has 4-colour palette.

Table 8-7: The series of experiments to show the tendency.

NodesNumber	UsedNodesNumber	Variant1 size	Variant2 Size
512	201	5632	3198
1024	215	12228	3648
2048	175	26642	3500
4096	124	53248	2814

The Variant1 size was calculated according to the formula 7.1. That is the reason why was chosen the second variant of the storing of the dictionary.

8.5 Storing the final dictionary in the compressed file

According to the previous paragraph the final dictionary has structure of a collection of sequences. So the first information, which is necessary to read the final dictionary, is the number of sequences (in future it will be called as *dictionary length*) and lengths of each sequence. As far as the number of sequences is about several hundreds they will be to encoded by Huffman coding. The process of storing the final dictionary in the compressed file can be divided into several stages, shown below:

1. Outputting the length of the dictionary (the number of initial dictionary elements after two stages of pruning).
2. Calculate length statistics for all lengths of the final dictionary's sequences.
3. Create a Huffman tree for the lengths according to lengths statistics.
4. Store the Huffman tree structure in the compressed file.
5. Output elements of the sequences.

The first point does not need any kind of explanations. This number is required by decoder to know exactly the number of the elements in the dictionary.

The second stage consists of two sub-stages. At the first sub-stage it is necessary to find the maximum length from all sentences from the final dictionary. The second is to calculate how often each length appeared in the final dictionary.

The third stage is related to the fourth one because the Huffman tree will be stored in the same manner as decompressor restores it. Before creating of the Huffman tree the compressor process the *scaling* of the probabilities. The process scheme is shown in the Figure 8-5.

```
MaxCount ← FindMaxCount(LengthCounts)
MaxCount ← MaxCount / 255
If(MaxCount = 0)
  MaxCount ← 1

For(i = 0; i < MaxLength + 1; i++ )
{
  If(LengthCount[i] / MaxCount = 0 && LengthCount[i]!=0)
    LengthCount[i] ← 1
  Else
    LengthCount[i] ← LengthCount[i]/MaxCount
}
```

Figure 8-5: Scaling of the length statistics

In the first step of scaling, the program finds the maximum count from all lengths counts. Then program divides the count on 255. The resulting number is stored into to the variable *MaxCount.* After all, the counts in the array of length counts are replaced by the quotient $\frac{LengthCount[i]}{MaxCount}$. This division guarantees that all count values now are less than 256. It allows us to describe each count by 8 bits ($\log_2 256 = 8$). The Huffman tree is creating now according to the modified counts.
Description of the Huffman tree in the compressed file is also very simple. The principals of this description are shown in the Figure 8-6.

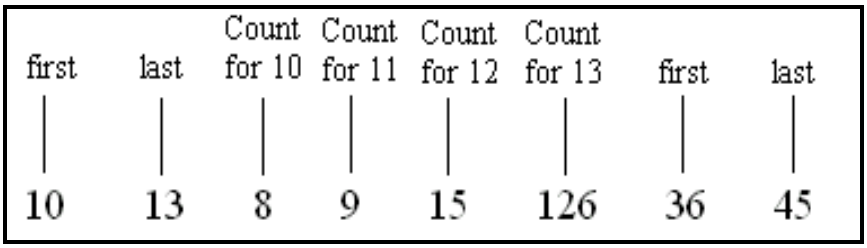


Figure 8-6: Example of Huffman tree structure storing in the compressed file.

To reduce the size of the Huffman tree structure, the information about counts is stored as triples: the first number is the lowest bound of the interval of counts, the last number is the highest bound of the interval, and the sequence of counts. All counts that are outside of the intervals have value of 0.

And when the lengths are outputted, it is time to start the last stage of the dictionary-storing process: the output of the dictionary elements. As far as the dictionary consists of sequences of symbols, it would be useful to represent all sequences as a big sequence and compress it by LZSS algorithm. The main idea of the LZSS coding of the dictionary is shown in the Figure 8-7.

```
CurrentPosition ← 1
CurrentElement ← 1
While(CurrentElement ≤ DictionaryLength)
{
  Symbol ← ReadSymbol(CurrentElement, CurrentPosition)
  EncodeLZSS(Symbol)
  CurrentPosition++
  If(CurrentPosition ≥ Length(CurrentElement))
  {
    CurrentPosition←1
    CurrentElement++
  }
}
```

Figure 8-7: pseudocode of the procedure of LZSS encoding of the semi-adaptive LZW dictionary.

Here the *CurrentPosition* is an offset from the beginning of the current sequence and *CurrentElement* is the number of sequence.
The general scheme of storing the dictionary to compressed file is shown in the Figure 8-8.

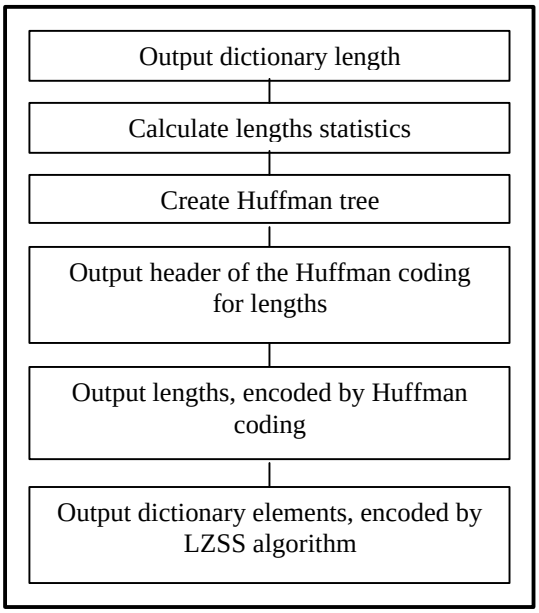


Figure 8-8: Scheme of final dictionary storing process

8.6 The output of compressed information

When all necessary information is outputted, then it is time to compress the input image. This algorithm is two-stages process:

1. Calculation of final dictionary elements statistics.
2. Outputting of the compressed information accordingly to the calculated statistics.

The idea of first stage is to find the modified cover of the input image according to the final dictionary elements. As was said before during first stage the first variant of the cover is created. But during the second stage some of the elements from the cover are deleted. In the compressed file they will be replaced by a new sequence of codes. The final dictionary does not have tree structure, but some of relations are kept. The structure of the final dictionary can be described in the following manner: each element has a *parent* and so-called *tail*: a sequence of symbols, which belonged to all initial dictionary elements that were between the parent and the current node. Let us consider the example from chapter 8.1. As follows from the chapter 8.2 in Figure 8-9 is shown the LZW tree after the first stage of pruning. In the Figure 8-3 is shown the original LZW tree for the example.

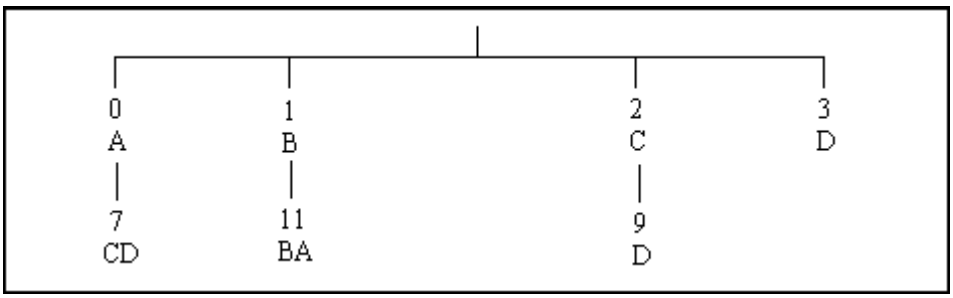


Figure 8-9: LZW tree for input string **"***AAABBACDBBAACDA***"** after first stage of pruning**.**

From the Figure 8-4 is clear that for the element with index 7 the parent is the element with index 0 and tail is the sequence "CD". In other words $7 = 0 + "CD"$, if code "*7*" will be pruned during second stage than it will be replaced by its parent (0) and code for its tail (code for "C" is

2, code for “D” is 3). The first stage is based on the same principles. If an element from the cover of the input image is not in the final dictionary then its parent and its tail replace it. The statistics are calculated in this case for the parent code and codes of symbols from tail. The pseudocode of the first stage is shown in the Figure 8-10.

```
String ← ReadSymbol(InputImage)
While(Not end of input stream)
{
  Symbol ← ReadSymbol(InputImage)
  if(String + Symbol ∈ InitialDictionary)
  {
     String ← String + Symbol
  }
  else
  {
     if(String ∈ FinalDictionary)
     {
       Output(CodeOf(String))
     }
     else
     {
       Output(ParentOf(String))
       Output(TailOf(String))
     }
  }
  String ← Symbol
}
```

Figure 8-10: Pseudocode of the encoding process in semi-adaptive algorithm.

In the first stage of semi-adaptive algorithm the function *Output* means calculating of statistics of elements, *InitialDictionary* is initial LZW dictionary; *FinalDictionary* is the initial dictionary after pruning.

When the first stage of outputting is over the second stage starts. The first stage, besides of the elements statistics gives also two values: the approximated number of bits required to encode the input image with the help of Huffman algorithm, and the number of bits required to encode the image without additional coding.

If the first number is less than the second, then the algorithm creates Huffman tree according to the calculated statistics. The structure of the Huffman tree is storing in the compressed file in the same manner that was described before. Then the algorithm outputs the header, that is necessary to deconstruct Huffman tree by decoder and then starts the algorithm, which is described in the Figure 8-9, but in this stage the function *Output* outputs Huffman codes
into the output bit stream.

Otherwise if the second number is less than the first one, the codes are outputting without any additional encoding. As far as the order of indexes of the elements in the final dictionary coincides with the order of the elements in the cover of the input image (in other words: an element with a bigger number can not appear during encoding of the image earlier than an element with a smaller number) the shrinking algorithm is applicable. But in the current algorithm the shrinking algorithm is modified. In first of all the current code bit length can decrease as well as it can increase. To determine such situation two reserved symbols “0” and “1” are used. It means that each outputed code increases by two before encoding (as it is impossible to use now codes “0” and “1”). Briefly the pseudocode of the algorithm of the simple outputting of a code is shown in the Figure 8-10.

```
Code ← Code +2
CodeBitLength ← GetCodeBitLength (Code)
NewCurrentCodeBitLength ← DetermineCurrentCodeBitLength(CodeBitLength)
If(NewCurrentCodeBitLength > CurrentCodeBitLength)
{
  Output (1)
  Output (NewCurrentCodeBitLength - CurrentCodeBitLength)
  CurrentCodeBitLength ← NewCurrentCodeBitLength
}
Else if (NewCurrentCodeBitLength < CurrentCodeBitLength)
{
  Output (0)
  Output (CurrentCodeBitLength - NewCurrentCodeBitLength)
  CurrentCodeBitLength ← NewCurrentCodeBitLength
}
Output (Code, CurrentCodeBitLength)
```

Figure 8-10: Brief description of outputting algorithm

Here *CodeBitLength* is the number of bits that is necessary to encode *Code* (in other words it is the value $\log_2 Code$). *NewCurrentCodeBitLength* and *CurrentCodeBits*, they are the values that define the code bit length. So, if due to the new coming code the algorithm decides that is better to increase or to decrease the *CurrentCodeBitLength* now, it outputs ”1” or “0” and the difference between them.

9. Experiments

We compress a set of topographic map images originating from the NLS topographic database [10]. The first image set contains of binary images and the second set of greyscale images. The set of experiments was provided to receive information about the previously described adaptive (modified shrinking method) and semi-adaptive (LZWsem) algorithms behaviour in the case of block-segmentation. For the experiments we had taken five binary images with size 1000×1000 pixels, and 4 greyscale images with size 1024×1024 pixels. During the experiments the block segmentation for each image was passed.

The binary images were processed the block-segmentation with sizes: 50×50, 100×100, 200×200, 250×250, 500×500 and 1000×1000 (the whole image compression). For comparison were taken from one side such widely known algorithms such as PNG, GIF and TIFF G4. The results of the compression of the binary images are in the Table 9-1 and Table 9-2. To get the average results of LZWsem compression for each image, we calculated an average of block-segmentation values for each image (see the Table 9-3). The dependency between average bit rate and the size of block is shown in the Figure 9-1, where the X-axis is the size of block and the Y-axis is the bit ratio.

For the greyscale images was processed the block-segmentation with next sizes: 64×64, 128×128, 256×256, 512×512 and 1024×1024 pixels in one block. The results of the series for greyscale images are in the Table 9-3 and 9-4. In the Figure 9-2 is placed the curve line of dependency between the average bit rate (average ratio between all greyscale images for one fixed size of block) and the block size.

Table 9-1: Compression results (bytes) for the set of binary images

Adaptive LZW (modified shrinking algorithm)					
	Image 1	Image 2	Image 3	Image 4	Image 5
50×50	53932	34501	48361	31418	63424
100×100	45525	26466	40353	22702	55936
200×200	41303	22298	36304	17439	51393
250×250	40488	21359	35585	16644	50790
500×500	38528	20136	34371	14183	47868
1000×1000	37625	19588	33282	12795	46678
Semi-adaptive LZW					
	Image 1	Image 2	Image 3	Image 4	Image 5
50×50	38352	19799	31233	14989	50790
100×100	37262	20747	31431	14383	48077
200×200	37449	20482	31737	13511	46026
250×250	38035	20813	32419	13802	46541
500×500	37425	20960	31175	13654	46369
1000×1000	37983	20801	30832	14289	46191

Table 9-2: Result of testing images compression by widely-known image compressors

Standard compression programs					
	Image 1	Image 2	Image 3	Image 4	Image 5
GIF	39303	20315	34641	13658	49168
PNG	36076	19372	34602	12625	45012
TIFF G4	24402	11316	19138	5212	29606
JBIG	14616	4763	10159	3274	18263

Table 9-3: Average results of compression of binary images (in *bpp*) (average in block-decomposition).

	LZWsem	GIF	PNG	TIFF G4	JBIG
Image 1	0.0377	0.0393	0.0360	0.0244	0.015
Image 2	0.0206	0.0203	0.0194	0.0113	0.005
Image 3	0.0315	0.0346	0.0346	0.0191	0.010
Image 4	0.0141	0.0137	0.0126	0.0052	0.003
Image 5	0.0473	0.0492	0.0450	0.0296	0.018

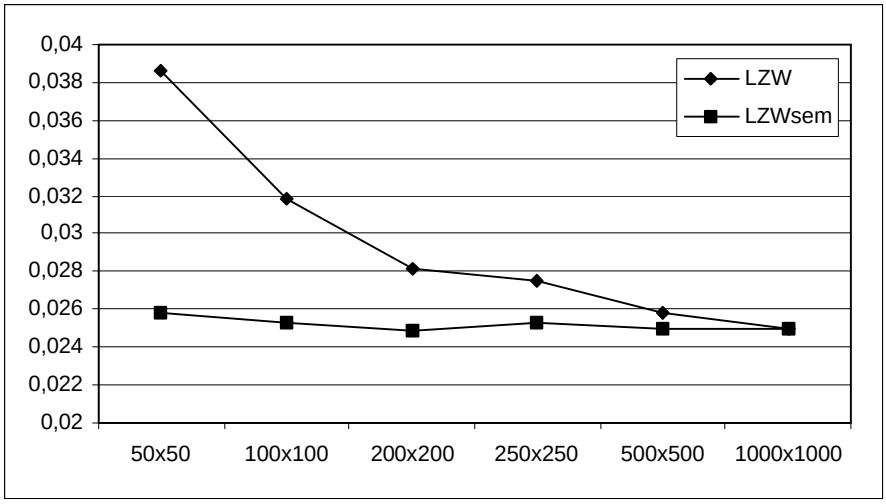


Figure 9-1 The curve line of dependency between the average bit ratios of LZWsem, modified adaptive LZW and size of block in the set of binary images.

Table 9-4: Compression results (bytes) for the set of greyscale images

Adaptive LZW(modified shrinking algorithm)				
	Image 6	Image 7	Image 8	Image 9
64×64	139421	326599	426650	356340
128×128	132702	313141	412271	341094
256×256	128179	306006	400110	331738
512×512	128092	305550	394732	328488
1024×1024	126714	312741	387807	326430
Semi-adaptive LZW				
	Image 6	Image 7	Image 8	Image 9
64×64	135976	325742	402940	341512
128×128	132984	325527	403132	335926
256×256	131507	311625	405255	335563
512×512	132484	310974	405182	335119
1024×1024	126714	312741	387708	326430

Table 9-5: The compression results for the set of greyscale images, compressed by standard compressor

Standard compression programs				
	Image 6	Image 7	Image 8	Image 9
GIF	138989	334066	433240	361842
PNG	152867	337140	423839	360734
TIFF Deflate	137840	332262	430912	359938
JBIG	14616	4763	10159	18263

Table 9-6: Average results of compression of greyscale images (in *bpp*) (average in block-decomposition).

	LZWsem	GIF	PNG	TIFF Deflate
Image 6	0.13	0.13	0.15	0.13
Image 7	0.30	0.32	0.32	0.32
Image 8	0.38	0.41	0.40	0.41
Image 9	0.22	0.35	0.34	0.34

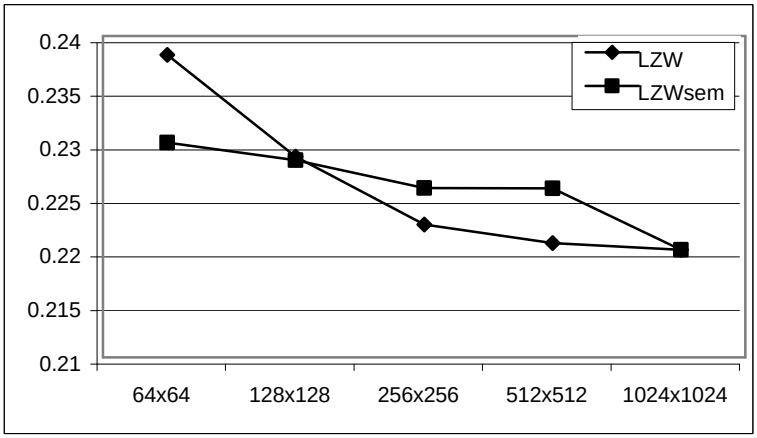


Figure 9-2: The curve line of dependency between the average bit ratios of LZWsem, modified adaptive LZW and size of block in the set of greyscale images.

10. Conclusions

In this thesis we have studied various modelling schemes and dictionary-based methods for map images compression. The details of existing methods were considered and new semi-adaptive method and modified shrinking method, which are based on the LZW compression, were introduced. Also the image tiling (block-segmentation) for dictionary-based methods was implemented. The proposed methods as well as GIF, PNG, TIFF and JBIG (for binary images only) methods were applied to the set of test images.

The empirical results show us that the decreasing of the block size will result in a rapid increase of the resulting code size for the pure adaptive methods such as the modified shrinking method. Also these results show us that for the semi-adaptive method the decrease of the block size has no affection to the resulting code size at all. So from the experiments it follows that on average an increase in the number of blocks by four times leads to an increase the compressed file size by 2-9 % only.

From all compression ratios, which were showed us by the proposed methods for the set of binary images, JBIG has the best one, TIFF G4 took the second place, adaptive and semi-adaptive methods have the same performance as GIF and PNG compressors. And for the set of greyscale image, they showed us the comparable performance with GIF, PNG and TIFF compressors.

References

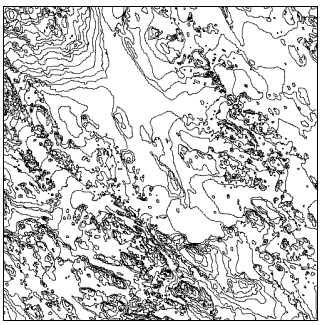
1. Nelson M., ”*The data compression book*”, IDG Books Worldwide, Inc., 1995.
2. Bell T., Cleary J., Witten I.H., “Modelling for text compression”, *ACM Computing Surveys*, Vol.21, No.4 (Dec.1989), pp.557-591.
3. Fränti P. “*Image Compression*”, Lecture Notes, Department of Computer Science, University of Joensuu, 2000, (http://cs.joensuu.fi/pages/franti/comp/comp.doc)
4. Deutsch P., “*DEFLATE Compressed Data Format Specification version 1.3*”, 1996, (ftp://ftp.nic.it/rfc/rfc1951.txt).
5. PKWARE Inc, “*ZIP File Format Specification*”, Version 4.5, Revised: 11/01/2001, (http://www.pkware.com/support/appnote.html)
6. Massachusetts Institute of Technology (MIT), “*PNG (Portable Network Graphics) Specification*”, 1996. (http://www.w3.org/TR/REC-png.html).
7. Smith S.W., “*The Scientist and Engineer's Guide to Digital Signal Processing*”, California Technical Publishing, (http://athena.vvsu.ru/carina/dsp/dspguide/datacomp.htm).
8. Fränti P., Ageenko E., Gröhn S., ”*Storage of multi-component digital maps using JBIG2 image compression standard*”, Research report A-2001-2, Department of Computer Science, University of Joensuu, 2001.
9. Fränti P., Ageenko E., Kopylov P., Gröhn S., Berger F., ”*Compression of map images for real-time applications*”, Research report A-2001-1, Department of Computer Science, University of Joensuu, 2001.
10. National Land Survey of Finland, Opastinsilta 12 C, P.O.Box 84, 00521 Helsinki, Finland. (http://www.nls.fi/index_e.html)
11. Ziv J., and Lempel A., “*A universal algorithm for sequential data compression*”, *IEEE Transactions on Information Theory*, Volume 23, Number 3, May 1977, pages 337-343.
12. Ziv, J., and Lempel A., “*Compression of individual sequences via variable-ratecoding*”, *IEEE Transactions on Information Theory*, Volume 24, Number 5, September 1978, pages 530-536.
13. Welch, Terry, “*A Technique for High-Performance Data Compression*”, *IEEE Computer*, Volume 17, Number 6, June 1984, pages 8-19.
14. Witten I., Neal R. and Cleary J., “*Arithmetic Coding for Data Compression*” *Communications of the ACM*, Volume 30, Number 6, June 1987, pages 520-540.
15. Huffman D., “*A method for the construction of minimum-redundancy codes*,” *Proceedings of the IRE*, Volume 40, Number 9, September 1952, pages 1098-1101.

APPENDIX: THE TEST SETS.

A. The test images for the semi-adaptive LZW compressor,

These images were mentioned in the paragraph 8.4. All images are maps. The map's size is written below each image.

Vantaa.pgm
800×800 pixels



zzz.pbm
500×500 pixels

v11.pgm
100×100 pixels

suomi2-q.pgm
400×600 pixels

B. Binary maps

These binary images were used in the set of experiments with the semi-adaptive LZW method. All of them are maps with size 1000×1000 pixels.

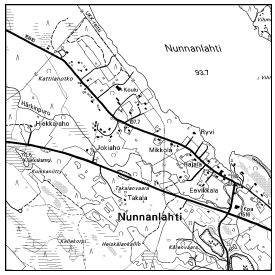


Image 1

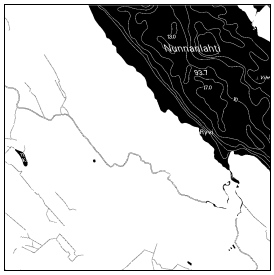


Image 2

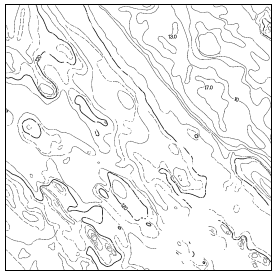


Image 3

Image 4

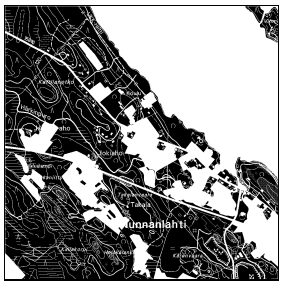


Image 5

C. Greyscale maps

As the binary images, these greyscale maps also were used in the set of experiments with the semi-adaptive LZW method. All of them are maps with size 1024×1024 pixels.

Image 6

Image 7

Image 8

Image 9