

Assignment 7 / Data Compression

Kenan Esau

April 2001

Tutor: Mr. Schmidt

Course: M.Sc Distributed Systems Engineering

Lecturer: Mr. Owens

Contents

1	Introduction	3
2	Question 1 – Compression Ratios	3
2.1	Compression Ratios for Text	3
2.2	Compression Ratios for Source Code	4
2.3	Compression Ratios for Binary Files	6
2.4	Conclusions	7
3	Question 2 – Compression Speed	8
3.1	Compression Speed for Text	8
3.2	Compression Speed for Source Code	8
3.3	Compression Speed for Binary Files	9
3.4	Conclusions	10
4	Question 3 – Compression Ratio and Compression Speed	10
5	Question 4 – Compression Ratio and Size of Dictionary	11
6	Question 5 – The LZ77 Algorithm	12
7	Question 6 – The LZ78 Algorithm	15
8	Question 7 – The Adaptive Huffman Algorithm	17
9	Question 8 – LZ78/Yet another Example	19
10	Conclusions	20

1 Introduction

This assignment tries to answer the questions of the workshop “Coding for Data Compression, Data Security, and Error Control”. Different loss-less compression algorithms are compared due to their performance (compression ratio) and speed. I will try to relate the results of the experiments to the different specialties of the different algorithms.

Three different programs are used:

1. GZip – which is an implementation of LZ77
2. Compress – which uses LZW
3. Compact – which implements adaptive Huffman coding.

2 Question 1 – Compression Ratios

The compression ratios of all three programs are measured with different file types and different file sizes.

2.1 Compression Ratios for Text

As you can see in figure 1, the compression ratios for very small files are very low, but it is increasing very rapidly with the size of the files. Compact has the lowest maximum compression ratio and it reaches this value at very low file sizes of about 10-20 Kbytes. Compress reaches the second best values for compression ratio. It reaches its maximum value much slower than the other two programs. GZip reaches its maximum compression ratio for text very fast (< 100 Kbytes).

The figure shows a straight line for compact for file sizes over 20 Kbytes. This is due to the fact that compact uses Huffman encoding. Huffman encoding is based on the probability distribution of different characters in human language (see section 8). Shorter codes are used for more probable character.

2.2 Compression Ratios for Source Code

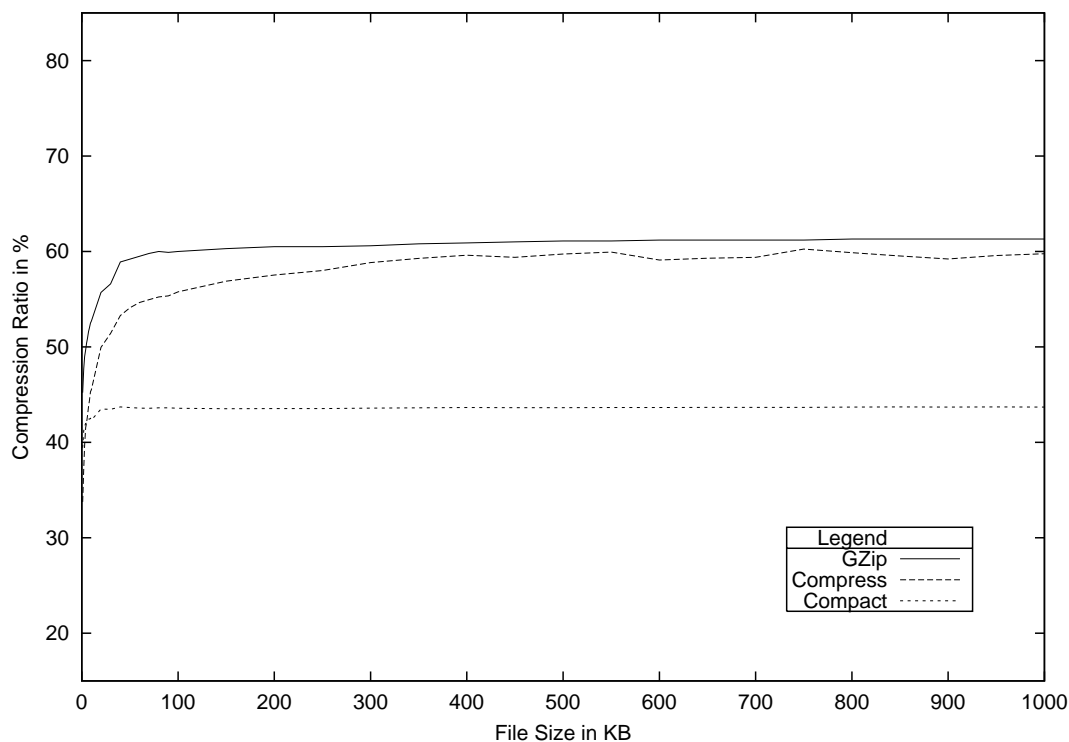


Figure 1: *Compression Ratios for Text*

2.2 Compression Ratios for Source Code

The compression ratios for source code are much better than for any other file type since source code contains the highest redundancy. For very large file sizes GZip reaches compression ratios beyond 80%. As you can see in figure 2, the compression ratio for very large file sizes increases for all of the three tested programs. It is interesting to see that the compression ratio for GZip begins to decrease for file sizes of about 80 Kbytes to 400 Kbytes until it starts to rise again for very large files.

GZip uses the LZ77 algorithm. For file sizes between 80 Kbytes and 400 Kbytes, this algorithm seems not to be able to find enough phrases in its “previously encoded” buffer (see [1]). Therefore it has to encode a lot of very short phrases or even single characters and a backward reference of the form `<start of the phrase in previously encoded buffer, phrase length, extension symbol>` is very long compared to a single character, thus the compression ratio decreases.

2.2 Compression Ratios for Source Code

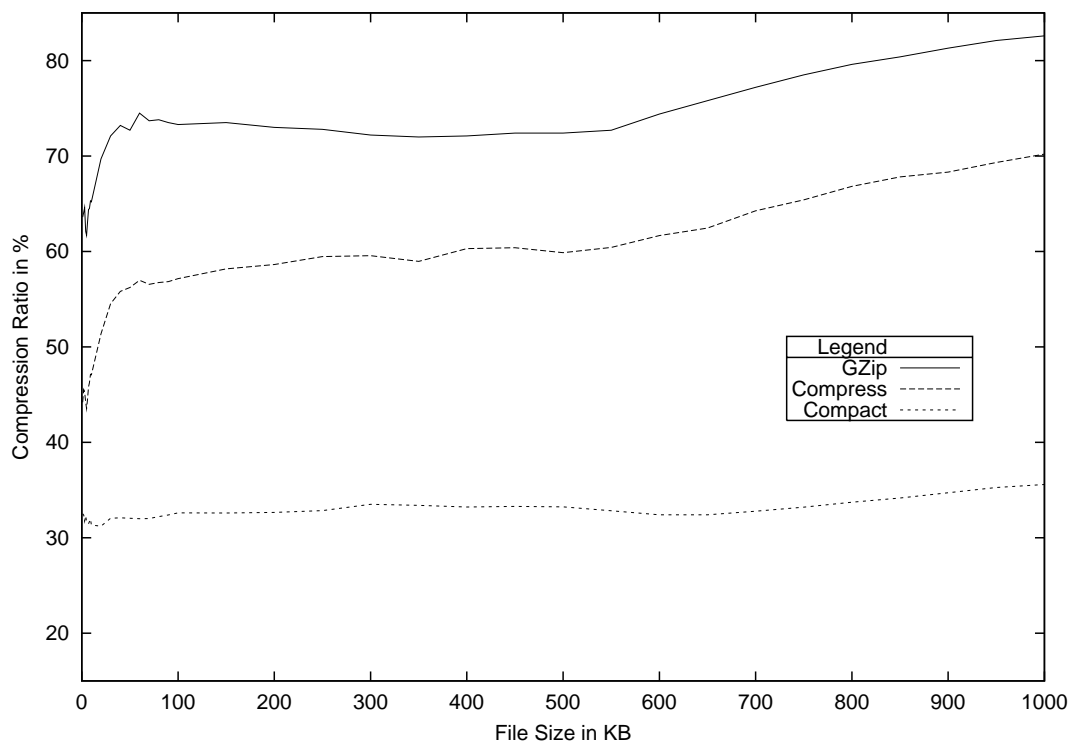


Figure 2: *Compression Ratios for Source Code*

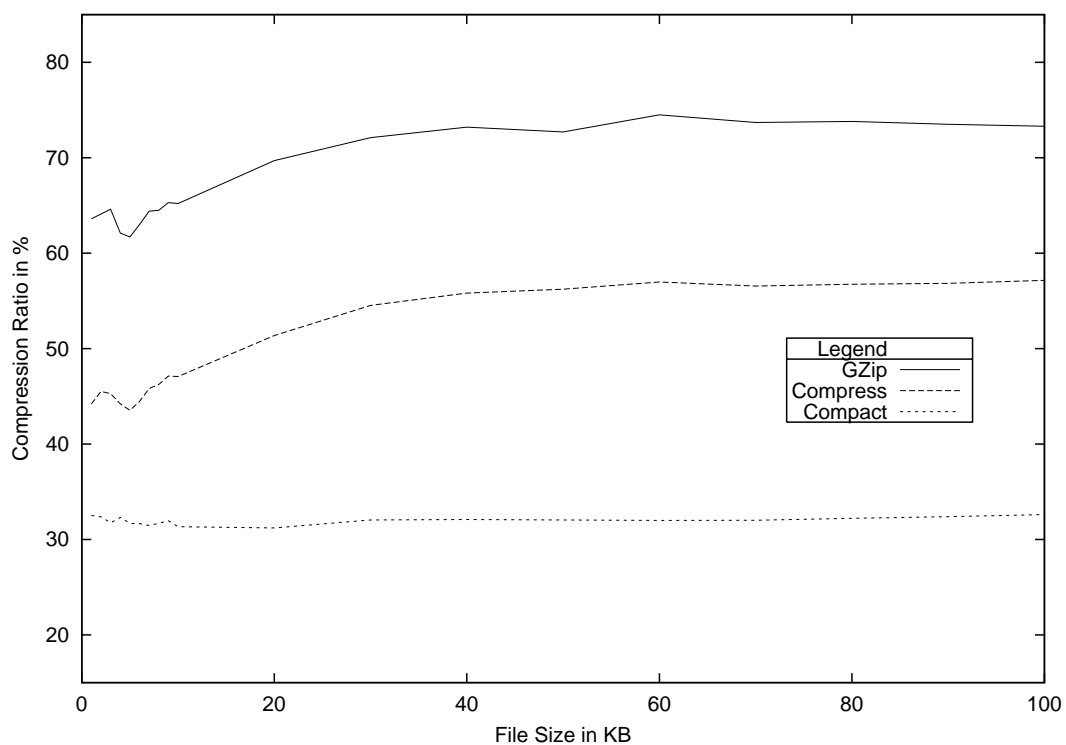


Figure 3: *Compression Ratios for Source Code*

2.3 Compression Ratios for Binary Files

For binary files, the compression ratios are the lowest compared to source code or text. That is since binary files contain the lowest redundancies. For file sizes about 400 Kbytes Compact's compression ratio even nearly drops to 15%. The compression ratios for all three compression methods decreases rapidly for file sizes between 20 Kbytes and 100 Kbytes.

In the figures 4 and 5 you can see that the compression ratios start at a very high values for the three programs and then decreases rapidly with increasing file size. You can also see that Compact is optimized for human language.

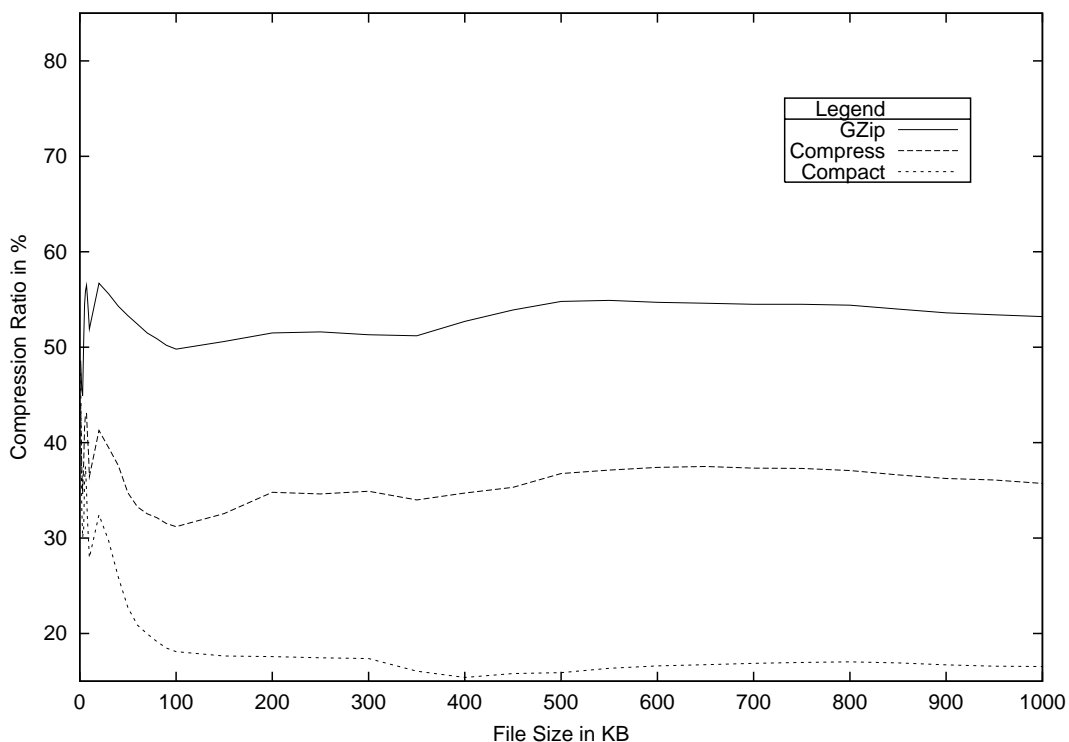


Figure 4: *Compression Ratios for binary Files*

2.4 Conclusions

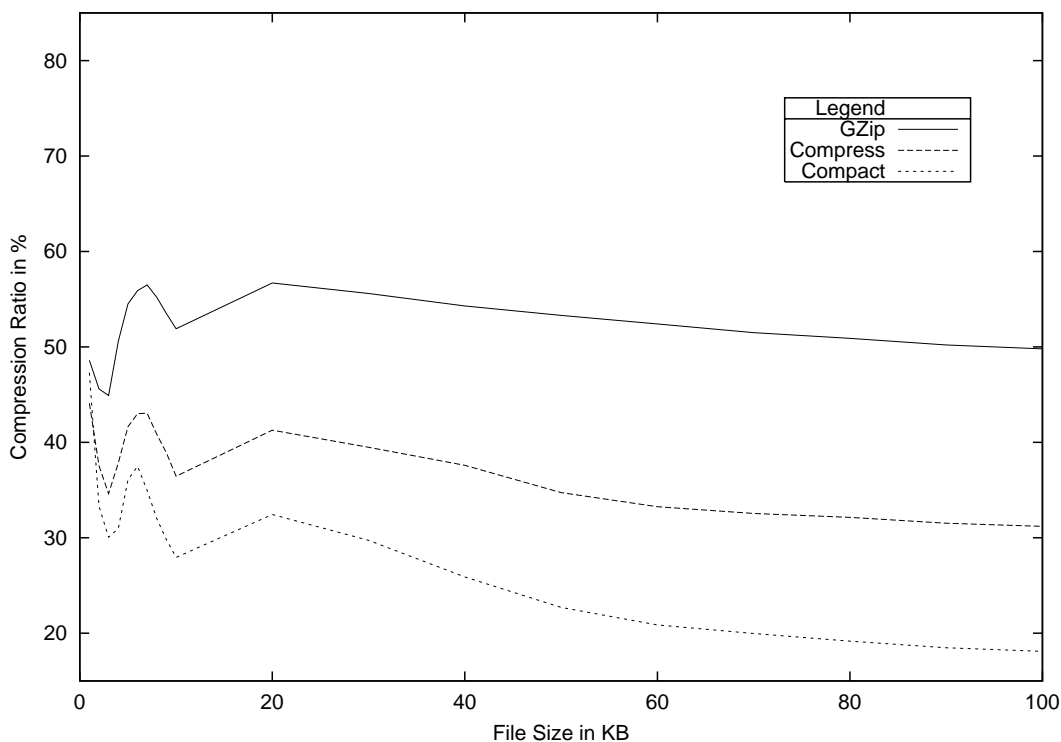


Figure 5: *Compression Ratios for binary Files*

2.4 Conclusions

The dictionary based methods achieve much higher compression rates than adaptive Huffman coding like it is used by compact. The LZW-method used by Compress achieves lower compression ratios than GZip (LZ77) since it has to discard its dictionary if it is full and after that it has to create a new one. This approach introduces much more overhead than with the LZ77 approach.

Additionally GZip is no pure LZ77 implementation. GZip uses the deflate algorithm which is a combination of LZ77 and Huffman coding (see man-page of GZip or [4]) and thus achieves much higher compression ratios than a pure LZ77-implementation. LZW does not send the extension symbol with the phrase pointer. The extension symbol is used as the first symbol of the next phrase [1]. This should result in improved compression.

3 Question 2 – Compression Speed

This section evaluates the results of different compression programs due to their speed. In this section I will try to show how the file size relates to the time needed to compress it.

3.1 Compression Speed for Text

The relationship between compression speed and file size is linear. The peak at the file size of 500 Kbytes for GZip can be explained with measurement inaccuracy and possible interaction of the operating system. Compact is the slowest of the three programs for text files, GZip is the second slowest and Compress is the fastest. GZip is slow compared to Compress but it achieves much higher compression ratios.

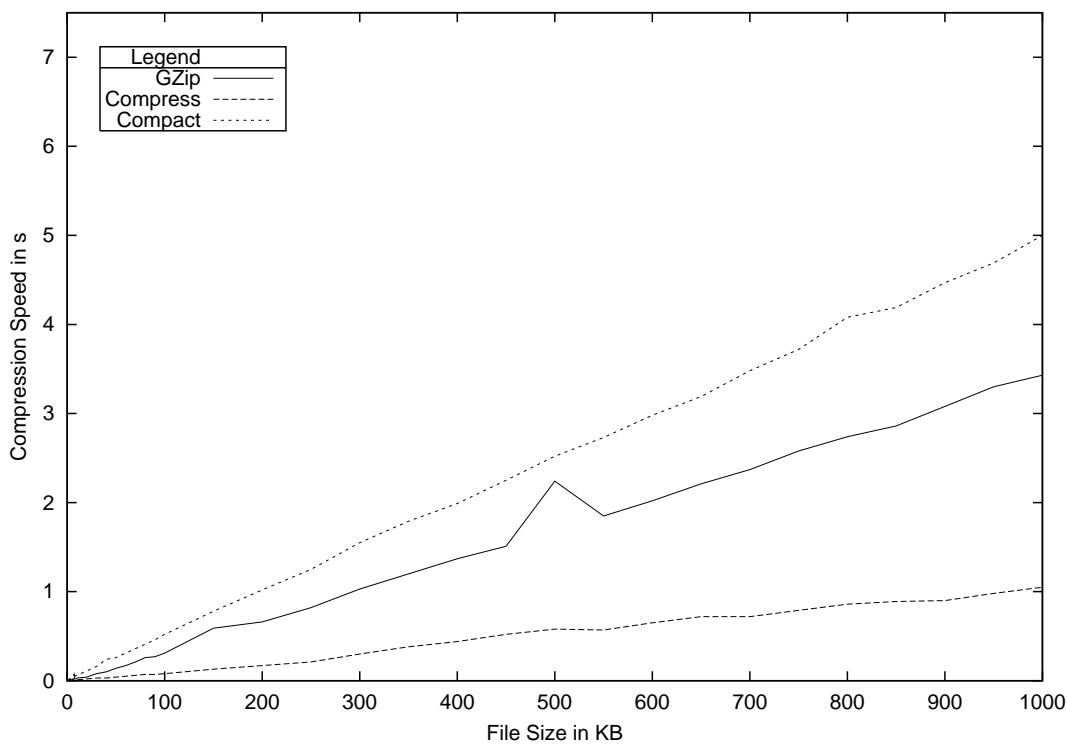


Figure 6: *Compression Speed for Textfiles*

3.2 Compression Speed for Source Code

The results for compression of source code are very similar to those of plain text. Compact is the slowest of the three programs, GZip is the second slowest, and Compress is the fastest solution. The relationship is linear. All peaks can be explained by operating system activities. GZip is for source code over two times faster than for text compression (assuming equal file sizes).

3.3 Compression Speed for Binary Files

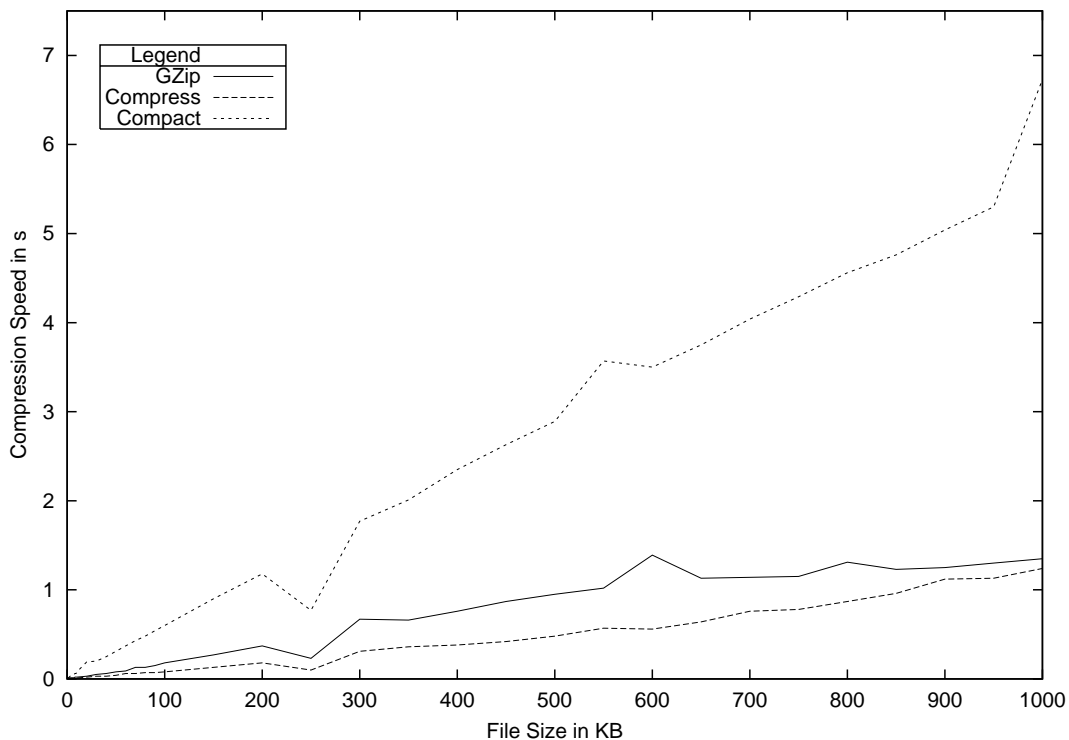


Figure 7: *Compression Speed for Source Code*

3.3 Compression Speed for Binary Files

The tendencies are the same as with the two former results. GZip is for binary files a little bit slower than for source code. Compress (LZW) seems to be unimpressed by binary files and the compression speed remains nearly the same as for the other file types. I think this is due to the fact that LZW was designed to be implemented in hardware and uses fixed sized pointers. It can be very fast if hashing is used.

3.4 Conclusions

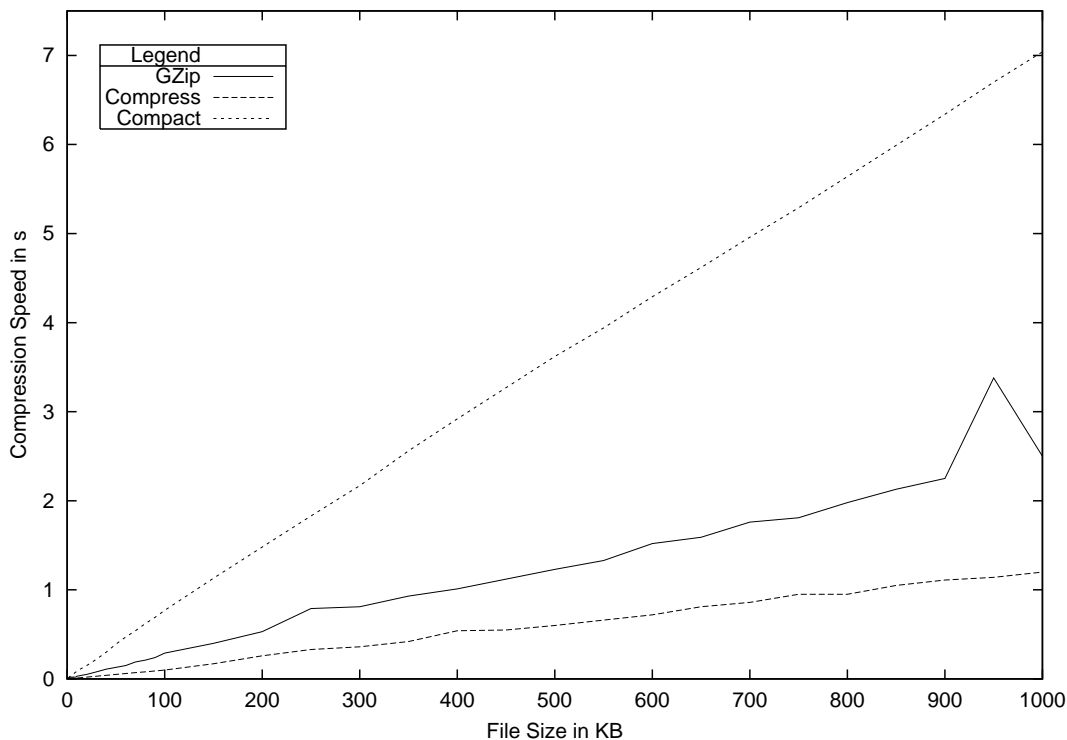


Figure 8: *Compression Speed for binary Files*

3.4 Conclusions

It is interesting to see that the speed of GZip varies widely between the different types of files. GZip is slow for text compression but source code and binary files are compressed a lot faster. But compression for source code is the fastest.

The speed of Compress (LZW-Algorithm) remains nearly the same for all three file types. Therefore compression speed for LZW seems to be independent of the type of data you are compressing.

4 Question 3 – Compression Ratio and Compression Speed

Figure 9 reveals no new secrets. As expected the time starts rising very fast with increasing compression ratio. At a compression ratio of about 60%, the time starts to increase very rapidly. So there is no point in trying to compress text files with a much higher ratio than 60% since it would take too long.

5 Question 4 – Compression Ratio and Size of Dictionary

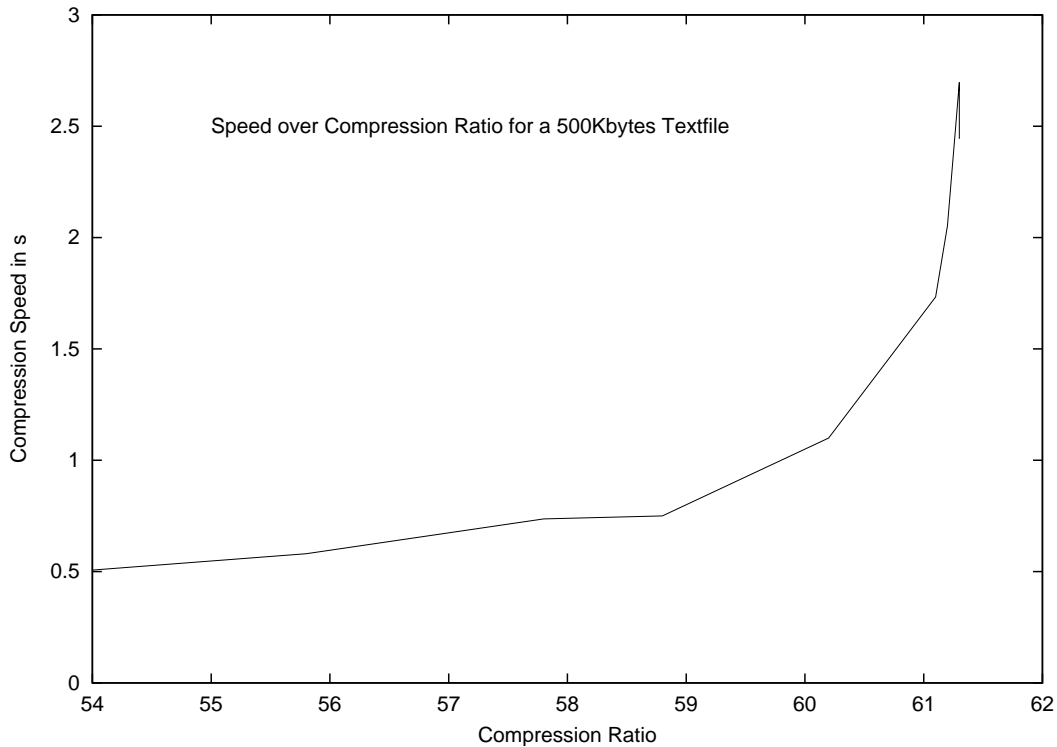


Figure 9: *Relationship of Compression Speed and Ratio*

5 Question 4 – Compression Ratio and Size of Dictionary

The compression ratio increases with the size of the dictionary. But similar as with compression ratio and time, there is a point where increasing the dictionary size does not result in an increase of the compression ration any more.

6 Question 5 – The LZ77 Algorithm

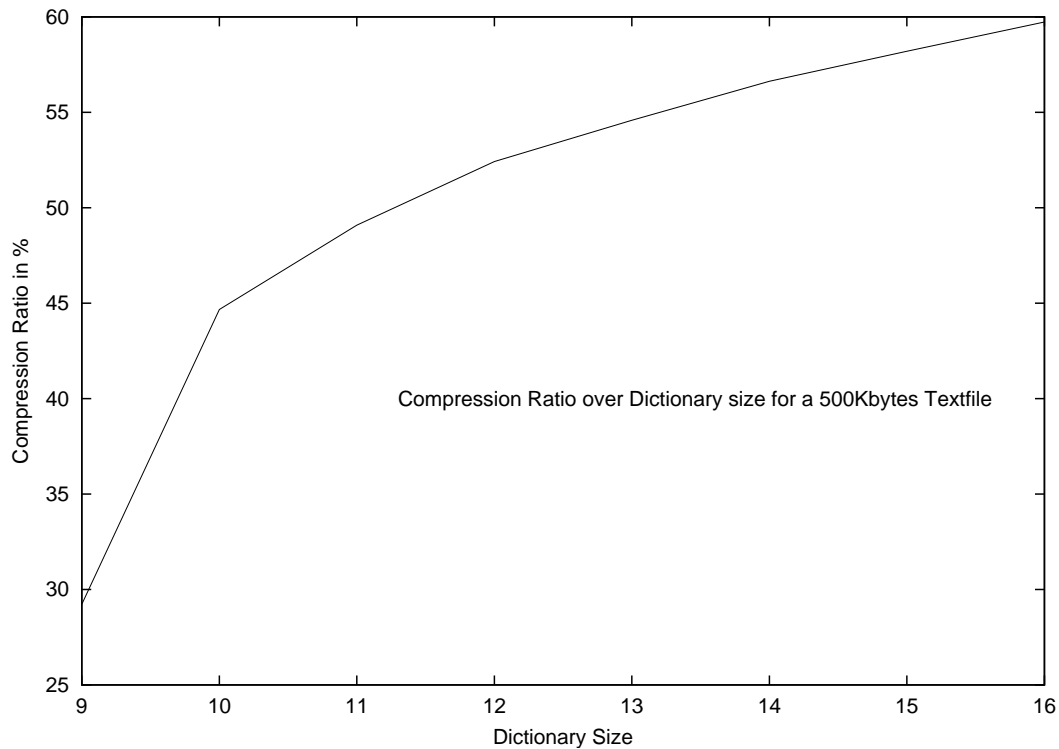


Figure 10: *Relationship of Compression Ratio and Dictionary size*

6 Question 5 – The LZ77 Algorithm

The LZ77 algorithm uses two buffers. One buffer which holds a certain amount of previously encoded strings. Here this buffer is referred to as “previously encoded buffer” – u. The second buffer holds the string which has to be compressed. This buffer is called “To be encoded buffer” – v. The exclusion of the last symbol in the v-buffer is essential since it has to be guaranteed that there is always an extension symbol [1].

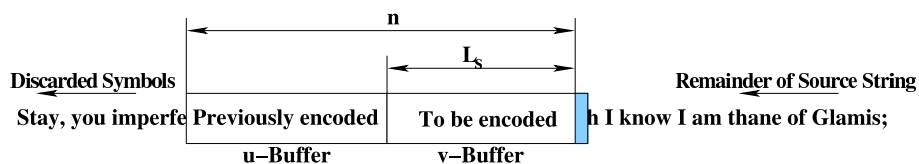


Figure 11: *Buffers of the LZ77 Algorithm*

Figure 12 shows the essential parts of the LZ77 algorithm. The first step is to initialize the u-buffer with a value (e.g. set the entire buffer to spaces – for text). After this the v-buffer has to be parsed for the longest possible match. During the first iteration our parser can not find a match unless the string which has to be compressed starts with spaces.

Now a code word of the form $\langle p, |\mu|, \sigma \rangle$ has to be composed. Where p is the position of the start of the match in the u-buffer, $|\mu|$ is the length of the match, and σ is the extension character. The extension character

6 Question 5 – The LZ77 Algorithm

σ is the next character occurring in the string which has to be encoded after the match found. A special thing about the matches is, that they can reach into the v-buffer.

The example below starts in the middle of a compression run since the start is quite boring. At the beginning of such a compression run – for the example shown below – you would have to write sequences like $\langle 0, 0, F \rangle$, $\langle 0, 0, a \rangle \dots$

The string I want to compress is a little piece from Macbeth: “Fair is foul, and foul is fair: Hover through the fog and filthy air.”

E.g. Assuming the u-buffer already contains:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a

And the v-buffer contains:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
n	d	␣	f	o	u	l	␣	i	s	␣	f	a	i	r

u-Buffer														v-Buffer																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	$\langle p, \mu, \sigma \rangle$
f	a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r	␣	$\langle 0, 0, n \rangle$
a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r		$\langle 0, 0, d \rangle$	
i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r			$\langle 6, 4, \text{␣} \rangle$	
␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r							...		

Table 2: Contents of u- and v-Buffer during Compression

If you try to produce table 2 according to the algorithm described by figure 12, the following steps have to be performed:

1. Parse buffer for longest match → Longest match is the character n since it does not appear in the u-buffer.
2. The code word is $\langle 0, 0, n \rangle$
3. The u-buffer and the v-buffer have to be shifted 1 character to the left (refer to line 2 of table 2)
4. Parse buffer for longest match → Longest match is the character d since it does not appear in the u-buffer.
5. The code word is $\langle 0, 0, d \rangle$

6 Question 5 – The LZ77 Algorithm

6. The u-buffer and the v-buffer have to be shifted 1 character to the left (refer to line 3 of table 2)
7. During the third iteration of our example the longest match found is the string `foul`.
8. Thus the code word is $\langle 6, 4, \sqcup \rangle$.
9. The u-buffer and the v-buffer have to be shifted 5 characters left (length of the string `foul` plus extension character) ...

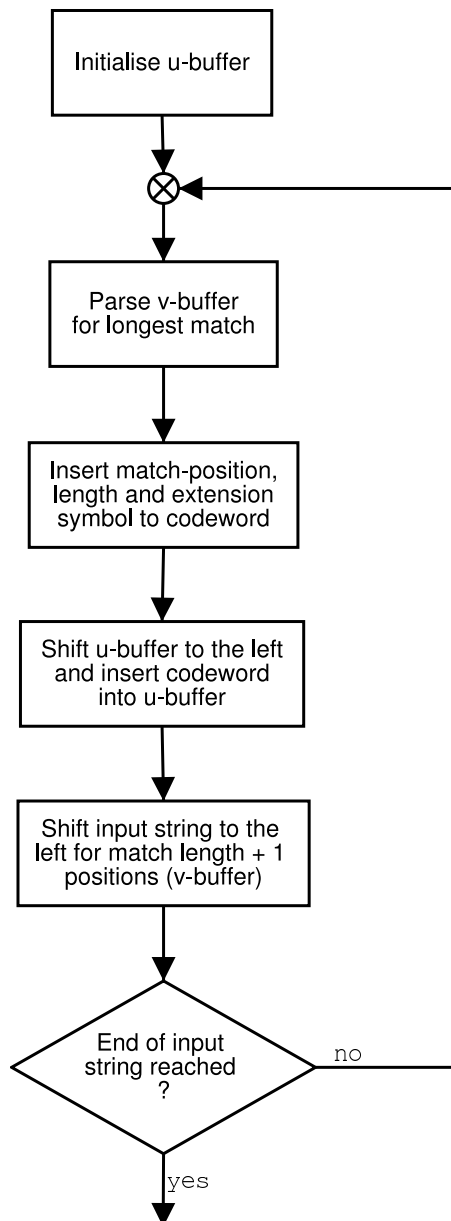


Figure 12: *Flowchart of the LZ77 Algorithm*

7 Question 6 – The LZ78 Algorithm

LZ78 works in a different manner as LZ77. It uses a fixed size dictionary which is filled up during the compression process. If the dictionary is full, the entries in the dictionary are discarded and a new dictionary is built from the scratch. LZ78 searches the dictionary for the longest match with the input string which has to be compressed. This match is called prefix or s since it is the prefix of the remainder of the input string.

If a prefix was found, a code-pair has to be encoded. This code pair is a reference to an earlier entry p in the dictionary plus the extension symbol σ .

This pair has to be encoded to form a single integer number k :

$$k = p|\Sigma| + I(\sigma)$$

Thus a mapping I from characters in the input alphabet Σ to the natural numbers $\{0, 1, \dots, |\Sigma| - 1\}$ has to be defined.

The integer number k which represents a pair consisting of a pointer to a string in the dictionary plus the extension symbol has to be sent (written to the output stream) as a bit word of the length $\lceil \lg(n|\Sigma|) \rceil$, where n is the position in the dictionary. The position 0 is always the empty string λ . Thus the first free position in the dictionary is $n = 1$.

The pointer to the prefix p and the extension symbol σ have to be written to the dictionary.

This process has to be continued until the dictionary is full or the string ends. If the dictionary is full it has to be reinitialized (discard all entries and initialize entry 0 with the empty string λ) and then the compression process can continue.

Let's have a look on the example used in the previous section: "Fair is foul, and foul is fair: Hover through the fog and filthy air."

This time I want to start the compression algorithm from the beginning. Each line in table 7 is a new iteration in the loop shown in figure 13.

The LZ78-compressed string uses 229 bits the uncompressed string takes up 248 bits (30 characters times 8 bits). In this a compression of 19 bits is achieved. This is since the string was very short. As you can see from table 7, the length of the binary string increases but the increase gets lower and lower due to the nature of the log-function. So you need only some more bits to encode much longer character-strings.

7 Question 6 – The LZ78 Algorithm

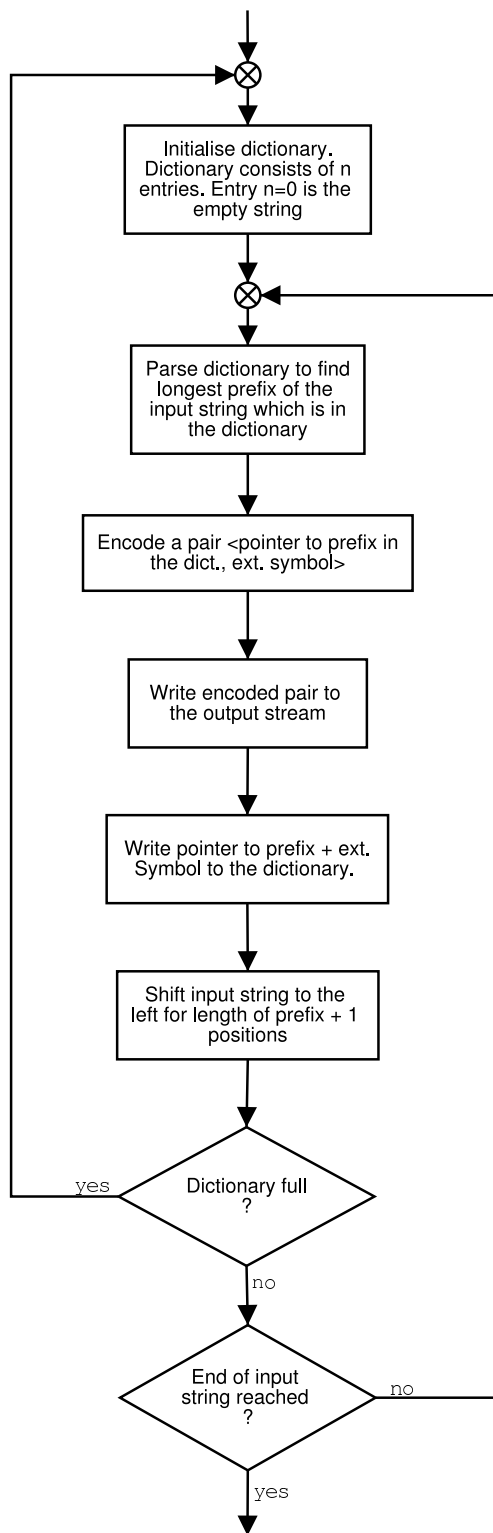


Figure 13: *Flowchart of the LZ78 Algorithm*

8 Question 7 – The Adaptive Huffman Algorithm

n	Phrase	Code Pair	$\lceil \lg(n \Sigma) \rceil$	Decimal Code k	Binary Code
1	F	$\langle 0, F \rangle$	8	70	01000110
2	a	$\langle 0, a \rangle$	9	97	001100001
3	i	$\langle 0, i \rangle$	10	105	0001101001
4	r	$\langle 0, r \rangle$	10	114	0001110010
5	␣	$\langle 0, \text{␣} \rangle$	11	32	00000100000
6	is	$\langle 3, s \rangle$	11	873	01101101001
7	␣f	$\langle 5, f \rangle$	11	1382	10101100110
8	o	$\langle 0, o \rangle$	11	111	00001101111
9	u	$\langle 0, u \rangle$	12	75	000001001011
10	l	$\langle 0, l \rangle$	12	108	000001101100
11	,	$\langle 0, , \rangle$	12	44	000000101100
12	␣a	$\langle 5, a \rangle$	12	1377	010101100001
13	n	$\langle 0, n \rangle$	12	110	000001101110
14	d	$\langle 0, d \rangle$	12	100	000001100100
15	␣fo	$\langle 7, o \rangle$	12	1903	011101101111
16	ul	$\langle 9, l \rangle$	12	2412	100101101100
17	␣i	$\langle 5, i \rangle$	13	1385	0010101101001
18	s	$\langle 0, s \rangle$	13	115	0000001110011
19	␣fa	$\langle 7, a \rangle$	13	1889	0011101100001
20	ir	$\langle 3, r \rangle$	13	882	0001101110010

Table 4: LZ78-Compression step by step

8 Question 7 – The Adaptive Huffman Algorithm

A disadvantage of the Huffman algorithm is its speed. It is very slow compared to the dictionary based methods (see section 3). The compression ratios achieved are also very low compared to the dictionary based methods (see section 2).

Huffman coding can give you an optimal code for the characters of an alphabet provided you know the probability distributions of the characters of the text you want to compress. Thus you have to send a table with each compressed file which tells the decompressor the codes of the character of the alphabet. It is also very expensive (computing time) to determine the probabilities of the different characters. If you assume to have every time the same probability distribution (e.g. assume you always want to compress English language text), you will achieve worse compression if you try to compress something with a different probability distribution.

The dictionary based methods do not need to send their dictionaries with

8 Question 7 – The Adaptive Huffman Algorithm

the compressed text since the dictionary can always be rebuilt provided you know the size of the dictionary used.

An “optimal compression method” would use a combination of both algorithms to achieve an optimal code for the output alphabet and to get good compression of reoccurring phrases. This approach is used in the deflate algorithm (see [5], [6]) which is used for example in the PNG picture format (see [2], [4], [3]).

9 Question 8 – LZ78/Yet another Example

This section shows an example of a LZ78-compression of the string: “the quick brown fox jumps over a lazy dog. the cat sat on the mat. the cat ate the mat. the cat sat on the hat”

You can find a discussion of the length of LZ78 encoded strings in section 7.

n	Phrase	Code Pair	n	Phrase	Code Pair
0	λ		33	\sqcup t	$\langle 4, t \rangle$
1	t	$\langle 0, t \rangle$	34	he	$\langle 2, e \rangle$
2	h	$\langle 0, h \rangle$	35	\sqcup c	$\langle 4, c \rangle$
3	e	$\langle 0, e \rangle$	36	at	$\langle 27, t \rangle$
4	\sqcup	$\langle 0, \sqcup \rangle$	37	\sqcup s	$\langle 4, s \rangle$
5	q	$\langle 0, q \rangle$	38	at \sqcup	$\langle 36, \sqcup \rangle$
6	u	$\langle 0, u \rangle$	39	on	$\langle 12, n \rangle$
7	i	$\langle 0, i \rangle$	40	\sqcup t	$\langle 4, t \rangle$
8	c	$\langle 0, c \rangle$	41	he \sqcup	$\langle 34, \sqcup \rangle$
9	k	$\langle 0, k \rangle$	42	ma	$\langle 19, a \rangle$
10	\sqcup b	$\langle 4, b \rangle$	43	t.	$\langle 1, . \rangle$
11	r	$\langle 0, r \rangle$	44	\sqcup th	$\langle 40, h \rangle$
12	o	$\langle 0, o \rangle$	45	e \sqcup	$\langle 3, \sqcup \rangle$
13	w	$\langle 5, w \rangle$	46	ca	$\langle 8, \sqcup \rangle$
14	n	$\langle 0, n \rangle$	47	t \sqcup	$\langle 1, \sqcup \rangle$
15	\sqcup f	$\langle 4, f \rangle$	48	ate	$\langle 36, e \rangle$
16	ox	$\langle 12, x \rangle$	49	\sqcup the	$\langle 44, e \rangle$
17	\sqcup j	$\langle 4, j \rangle$	50	\sqcup m	$\langle 4, m \rangle$
18	u	$\langle 0, u \rangle$	51	at.	$\langle 36, . \rangle$
19	m	$\langle 0, m \rangle$	52	\sqcup the \sqcup	$\langle 49, \sqcup \rangle$
20	p	$\langle 0, p \rangle$	53	cat	$\langle 46, t \rangle$
21	s	$\langle 0, s \rangle$	54	\sqcup sa	$\langle 37, a \rangle$
22	\sqcup o	$\langle 4, o \rangle$	55	t \sqcup o	$\langle 47, o \rangle$
23	v	$\langle 0, v \rangle$	56	n \sqcup	$\langle 14, \sqcup \rangle$
24	er	$\langle 3, r \rangle$	57	th	$\langle 1, h \rangle$
25	\sqcup a	$\langle 4, a \rangle$	58	\sqcup c	$\langle 4, c \rangle$
26	\sqcup l	$\langle 4, l \rangle$	59	at \sqcup s	$\langle 38, s \rangle$
27	a	$\langle 0, a \rangle$	60	at \sqcup o	$\langle 38, o \rangle$
28	z	$\langle 0, z \rangle$	61	n \sqcup t	$\langle 56, t \rangle$
29	y	$\langle 0, y \rangle$	62	he \sqcup t	$\langle 41, t \rangle$
30	\sqcup d	$\langle 4, d \rangle$	63	he \sqcup th	$\langle 62, h \rangle$
31	og	$\langle 12, g \rangle$	64	e \sqcup h	$\langle 45, h \rangle$
32	.	$\langle 0, . \rangle$	65	at.	$\langle 36, . \rangle$

Table 7: LZ78-Compression step by step

10 Conclusions

This assignment shows the relationship between compression ratio, compression time, file size/type and dictionary size. From the various diagrams you can see the different advantages of the three algorithms used and how those algorithms behave according to the files they have to compress.

The results of this assignment were as expected. This was a very uncommon assignment compared to the previous ones since it was more strict. There were a lot of questions which had to be answered. Thus there was absolutely no freedom in choosing a topic and very few freedom in doing discussions of a topic.

Due to the great amount of diagrams the effort for creating this assignment was very high. I personally do not like this style of assignment since the effort of writing it is very high but there are nearly no new things you can learn from it since the most conclusions were already drawn during the workshop. Documenting those conclusions brings nothing new.

References

- [1] Thomas Owens (Script) “Coding for Compression and Data Security”
- [2] PNG Development Group, “PNG Specification Version 1.2”
- [3] Kenan Esau “Assignment 6/Multimedia – The PNG-Format”
- [4] RFC-2083, “PNG-Format”
- [5] RFC-1950, “Zlib Specification”
- [6] RFC-1951, “Deflate Specification”