LOW COMPLEXITY PARAMETRIZED CODES FOR LZ77 COMPRESSION

Péter A. FELVÉGI

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
H–1111, Budapest, Goldmann György tér 3, Hungary
e-mail: petschy@avalon.aut.bme.hu

Received: Nov. 3, 2000

Abstract

In the LZ77 compressors family the compression ratio can be increased in two possible ways: first, by better parsing of the input data into <*distance, length*> pairs and <*literal*> characters, and second, by better encoding of the result of the parsing.

The parsing can be enhanced by increasing the size of the sliding window and by using sophisticated parser heuristics to decide which match to take and which to discard. This topic was studied by several people already and is beyond the scope of this paper.

The efficiency of coding depends on how good estimates we can give on the probabilities of the *distance*, *length* and *literal* values. The most widely used LZ77 derivatives use a semi-static approach with Huffman coding. This requires two passes over the input data and the transmission of the codetables along with the compressed data, too.

In this paper I investigate the case where constraints are on processing power and memory at the decompressor side. Having a certain parser, we want to code the <*distance, length*> pairs and <*literal*> characters as effectively as possible. Because of the constraints I omit arithmetic code and also omit Huffman code.

Codes for representing integers of an assumed distribution (with exponentially decaying overall behaviour) are already proposed by several researchers. These codes are simple, require no additional memory but can adapt only in a restricted way or not at all to the actual distribution of the values. Thus they only perform well if the actual distribution is 'near' to the assumed one.

I will present a new family of codes that keep the simplicity of the already known codes but can adapt better to the actual distribution through a few integer parameters. As the distribution is still assumed to have exponentially decaying tendency these codes perform well encoding the *distance* and *length* values, but usually are not suitable for the *literal* characters.

Keywords: LZ77, compression, source coding.

1. LZ77 Compression — Introduction

The family of LZ77 derived compression algorithms is rather long. A lot of improved variants have been proposed since the publication of the original algorithm [1] in 1977. Of these, some have focused on how to improve the parsing in terms of temporal and spatial complexity, the others dealt with the coding part of the algorithm. Nowadays the most widely used compression programs also employ some variation of the LZ77 algorithm (e.g.: *zip*, *gzip*, *arj*, *rar*).

The LZ77 algorithm is a dictionary-based compression method. This kind of methods uses the principle to replace substrings of the text with references to the dictionary. In the case of the LZ77 algorithm the dictionary is dynamic, i.e. it changes continuously as the input is processed. As the matter of fact the dictionary is the previously seen N characters of the text. References to the dictionary are made by a pair of values : <*distance, length*>.[1] The *distance* tells us how far back from the current position begins the substring and the *length* shows how many characters the substring consists of. Of course, there are cases when the substring at the current position is not in the dictionary, i.e. it did not exist before. In that very case a *literal* character is emitted[2] and the algorithm continues from the next position. The <*distance, length*> pairs and the *literal* characters are then coded. Surprisingly, a reasonable amount of compression can be achieved by storing these values with flat binary code only without paying attention to the actual distribution. Of course, the more sophisticated the coding part is the better the compression will be. The general block diagram of the compression process is shown in *Fig. 1*. The only thing that has not been mentioned yet is the coding scheme. The scheme tells the coder how to distinguish between literals and matches, what codes to use, etc. For an example refer to the test results on page 21 or see [3].

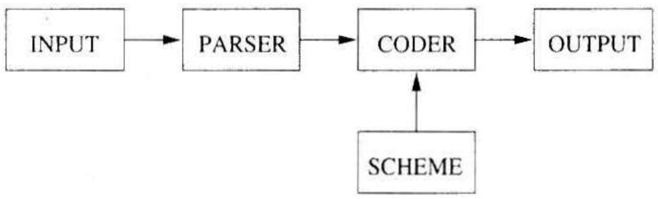

Fig. 1 General block diagram of the LZ77 compressor

1.1. Parsing

The process of dividing the text into <*distance, length*> pairs (references of substrings in the dictionary) and <*literal*> characters (that are not in the dictionary yet) is called parsing. There are several algorithms for effective parsing considering speed and memory requirements. The parsing also affects the compression ratio, because usually there are several matching substrings in the dictionary with the one

[1] The original LZ77 paper used triples of <*distance, length, literal*>. This is rather inefficient, so I am using here the approach of separating the references from the literals found in [2].

[2] the one at the current position

at the current position. The process of deciding which match to take is usually controlled by some heuristics. This topic has been extensively studied and is beyond the scope of this paper.[3]

1.2. Coding

After parsing, coding takes place. We want to achieve as good compression as possible so we must take the probability distributions of the *distance*, *length* and *literal* values into account and employ some sort of source coding. The above mentioned compression programs use Huffman coding in a semi-static manner: first they make frequency counts, then build the appropriate code for that distribution. This method usually gives quite good coding efficiency, although it has some drawbacks. First, one has to process the data twice and second, one must transmit the codetables, too.[4] If there are limitations in processing power and memory on the decompressor side (which is the assumption of the present investigation) one has to look for other kinds of codes.

The flat binary code works fine of course, but we can do better than that. We can benefit from the general observation that for the *distance* and *length* values the smaller values tend to be more frequent and the larger values seem to occur less frequently. For the *literal* values we cannot state anything similar, because the distribution depends more on the kind of data being compressed. Because of that, from this point we focus on the encoding of the *distance* and *length* values only.

The general observation of the distributions of the *distance* and *length* values leads to the conclusion that the distributions have an overall exponentially decaying tendency. Of course, this is just a rough estimate, there could be serious deviations for some kind of files[5], but for the majority of cases, it is good enough. There are codes already that might be used for coding values with this kind of distribution, I will discuss them in more detail after the decompression part.

1.3. LZ77 Decompression

The decompression is much simpler and faster than the compression, because no string matching is required (which is usually the most time consuming part of the compression). After decoding a *literal* character there is just a memory write; after decoding a <*distance*, *length*> pair there is just a *memcpy*().

The speed of the decompression is mainly determined by the speed of the decoder. Here we can see the asymmetric property of the LZ77 algorithm from the

[3]For the ones who are interested in how the parsing is made in practice I suggest starting with [3] and [4], about the heuristics one might read in [5] and [6] and find further references.

[4]In practice this is usually just a marginal overhead, although for small files it may become significant.

[5]See more about this later; the *kennedy.xls* file from the Canterbury Corpus is a nice example.

compression–decompression point of view. If the coder is reasonably fast[6], then decompression will be significantly faster than compression.

2. Simple Codes Already Known

There are already simple codes proposed by several people for assumed distributions. The common in these codes is that they consist of two parts: first a unary code, then a binary[7] code whose length is fixed or dependent on the value decoded from the unary part.

Some of the following codes originally started coding numbers with one instead of zero. For me starting from zero is more logical since things usually start from zero when one works with computers. Because of that I made all the codes start from zero.

2.1. Unary Code

This code is the simplest of all. To encode the number $i \geq 0$ one has to emit i ones and a zero bit at the end. Since we know from [7] that there is a relationship between the ideal coded length and the probability

$$l_i = -\log_2 p_i$$

we can calculate the probability distribution for which the code is optimal:

$$\begin{aligned} l_i &= i + 1, \\ p_i &= 2^{-l_i}, \\ p_i &= \frac{1}{2^{i+1}}. \end{aligned}$$

As it can be seen the distribution is exponentially decaying, though the decay is far too drastic for our current purpose.

2.2. Binary Code

Of course this code is known by everyone. To encode a number $0 \leq i < 2^N$ we use N bits. The implied probability distribution is

$$\begin{aligned} l_i &= N, \\ p_i &= \frac{1}{2^N}, \end{aligned}$$

[6]faster than the parser

[7]The δ code is an exception.

i.e. uniform.

All the following codes have the common property that their implied probability distributions lie between the binary exponential decay of the unary code and the uniform distribution of the binary code.

2.3. *Elias γ and δ Codes*

These codes were first described in [8]. The γ code consists of a unary and a binary part. The length of the binary part depends on the unary value. Let the unary value be u then the length of the binary part is $l_{bin} = u$ bits. According to [9] the implied probability distribution is

$$p_i = \frac{1}{2(i+1)^2}.$$

The δ code is similar, but there we have γ code instead of the unary code. This way the codes grow less rapidly which corresponds to a less drastic decay in the distribution. The implied probability distribution is

$$p_i = \frac{1}{2(i+1)(\log_2(i+1))^2}.$$

2.4. *Golomb and Rice Codes*

These codes still consist of a unary and a binary part, yet they have an additional integer parameter, b. Coding the value i is done in two steps according to [10]. First, write the quotient $q = \lfloor i/b \rfloor$ with unary code, then the remainder $i - qb$ on either $\lfloor \log_2 b \rfloor$ or $\lceil \log_2 b \rceil$ bits coded in binary. The codelength corresponding to the value i can be approximated from above with

$$l_{i,b} = 1 + q + \lceil \log_2 b \rceil,$$
$$l_{i,b} = 1 + \lfloor \frac{i}{b} \rfloor + \lceil \log_2 b \rceil,$$

thus the probability distribution is

$$p_{i,b} = \frac{1}{2^{1+q+\lceil \log_2 b \rceil}},$$
$$p_{i,b} = \frac{1}{2^{1+\lfloor \frac{i}{b} \rfloor+\lceil \log_2 b \rceil}}.$$

Rice codes have a restriction on parameter b to be an integer power of two: $b = 2^n,\ n \in Z^+$.

2.5. *Start-Step-Stop Codes*

Again, a unary and a binary part build the code, where the length of the binary part is given as $l_{bin} = start + step \cdot u$ where u is the unary coded value. The stop parameter makes it possible to omit the zero from the end of the unary code for the greatest values. This code was proposed in [4] and as it can be seen, it uses three integer parameters.

3. The New Parametrized Codes

Let us suppose that there are restrictions on the decompressor side in computing power and available memory. In this case the compression ratio can be enhanced in two ways. First, using a more sophisticated parser algorithm, since this increases the compressor complexity only and the decompression remains as fast as it was. Second, employing more effective codes in the coder. We will deal with this second means of enhancement in the rest of this paper.

Since we want to keep the complexity at a minimum, we omit arithmetic coding and also omit Huffman coding. We want codes that keep the simplicity of the above mentioned ones while they can approximate the actual probability distribution more precisely.

Generalizing the idea of the previously mentioned codes the code still consists of a unary and a binary part, yet the length of the binary part is an arbitrary function of the unary value

$$l_{bin} = f(u).$$

Let us assume that the probability distribution is exponentially decaying, from this we can calculate the ideal coded length:

$$\begin{aligned} p_i &= ax^{-\alpha i},\ i \geq 0, \\ l_i &= -\log_2 ax^{-\alpha i}, \\ l_i &= -(\log_2 a - \alpha i \log_2 x), \\ l_i &= \alpha i \log_2 x - \log_2 a, \end{aligned}$$

where a is chosen so that the distribution is normalized, $x > 1$ is the base of the exponent and $\alpha > 0$ controls the rate of the decay. After substituting the constant expressions with capital letters we have

$$l_i = Xi + A,$$

where

$$\begin{aligned} X &= \alpha \log_2 x, \\ A &= -\log_2 a, \end{aligned}$$

thus the ideal coded length is a linear function of the value i to be coded.

The ideal coded length is usually not integer, yet we have to deal with integer number of bits. Anyway, we know that the length should be a linear function so we choose $f(u)$ to be

$$f(u) = \lfloor ru + b \rfloor,$$

where r and b are nonnegative real numbers. The total number of bits needed then is

$$l = \lfloor 1 + u + ru + b \rfloor,$$
$$l = \lfloor u(1 + r) + (b + 1) \rfloor.$$

This is a linear function of u (inside the $\lfloor\ \rfloor$), just as we wanted it to be.

We want integer parameters, because dealing with real (in practice: rational) numbers is slower. Thus we write

$$f(u) = \left\lfloor \frac{P}{Q} u \right\rfloor + B,$$

where B, P and Q are integer parameters, ($B,\ P \geq 0$ and $Q \geq 1$).

We introduce a fourth parameter C to have control over the point where the rounding jumps to the next integer value. The final form of $f(u)$ is then

$$f(u) = \left\lfloor \frac{Pu + C}{Q} \right\rfloor + B.$$

The total number of bits needed is

$$l = 1 + u + \left\lfloor \frac{Pu + C}{Q} \right\rfloor + B.$$

As it can be seen the code can be described with four parameters $<B, C, P, Q>$. These parameters have to be optimized for the actual distribution, so we use the codes in a semi-static manner. This is not a problem, since it is done in the compressor, and the decompressor will need only the parameters, whose transmission is just a very small overhead.[8]

The only thing that remains is to find the connection between i and u. We have seen that l_i should be a linear function of i for an exponential distribution. But what we have now (neglecting the $\lfloor\ \rfloor$) is a linear function of u. If the correspondence between i and u is linear, then l will be a linear function of i also. Unfortunately that is not always the case depending on the parameters.

The relationship between u and i is

$$i < \sum_{j=0}^{u} 2^{f(j)}$$

[8]One byte is usually enough for a parameter, so the overhead totals 4 bytes.

and u is the smallest number for which the inequality holds. It is clear that the $u = g(i)$ function will only be linear if $f(j)$ is constant, and usually this is not the case.

Is it a real problem? Because of rounding, the relationship cannot be entirely linear anyway and the actual distributions are not exactly exponential either. As we will see from the test results the code performs quite well on real-world files.

Interestingly, a few of the above mentioned simple codes can be achieved with certain parameters:

- <0,0,0,1> is the unary code
- <0,0,1,1> is the gamma code
- <B,0,P,1> is the start-step-stop code with $start = B$, $step = P$ and $stop = \infty$
- <B,0,0,1> is the Rice code with parameter 2^B

It is clear that the proposed new code can perform better (or at least equally) than these codes because it can exploit all the freedom offered by its parameters.

4. Empirical results

4.1. Practical Considerations

For practical reasons I have made three variants of the code using two, three and all four parameters:

$$f(u) = Pu + B,$$
$$f(u) = \left\lfloor \frac{u + C}{Q} \right\rfloor + B,$$
$$f(u) = \left\lfloor \frac{Pu + C}{Q} \right\rfloor + B.$$

These will be referenced later as C2, C3 and C4 codes. Of course, the C4 code will always be the best (or at least equal), followed either by the C2 code or the C3 code depending on the rate of the decay of the actual distribution.

4.2. Optimizing the Parameters

As I have said before, the codes should be used in a semi-static manner, because parameters must be optimized for the actual distribution. I made exhaustive search with some heuristics based on the meaning of the parameters to find the optimum. There might be a more effective optimization algorithm for that purpose, but this is beyond the scope of the current investigation.[9]

[9] The optimization is not easy, because we do not have the function in analytical form, thus it is not differentiable, it is multivariate and we have to deal with integer numbers.

- **Parameter B:** I thought first that the codelength based on p_0 was a good estimate[10], but the tests showed that it was not usually the case. Because of that, B goes from 0 to a user defined maximum.[11]
- **Parameter C:** Goes from 0 to $Q - 1$ since above that the effect is the same as increasing B.
- **Parameter P and Q:** P goes from 0 to a user defined maximum.[12] Q goes from 1 to the same maximum as P. That maximum value affects in how fine granule can be the slope of $\frac{P}{Q}u$ altered. The finer the granule, the better the approximation, but this requires more loops in the optimizer, thus slows the process down.

4.3. The Coding Scheme Used

The coding scheme used in the tests was rather simple. The actual codes for the *length* and *distance* values differed only, everything else was the same to have fair comparisons. The scheme was the following:

- 1 bit decision
- 8 bits literal
- γ/δ/Huffman/C2/C3/C4 coded length
- binary coded the 7 lowest bits of the distance
- γ/δ/Huffman/C2/C3/C4 coded the remaining upper bits of the distance

4.4. Results on the Test Files

The tests run on a PC with Pentium-233MMX processor under linux. All the coders and other modules were written by myself; they are not optimized (yet) for speed, rather they help to build compression programs fast to test new ideas.

Two compression test suites were used: the **Calgary Corpus** and the **Canterbury Corpus** as they are widely used for compression benchmarks.

In *Figs. 2* and *3* the compression ratios are shown in percentage. The ratio is computed as

$$\text{ratio} = 100\% \cdot \frac{\text{compressed size}}{\text{original size}}$$

so the smaller the number, the better the compression. The meaning of the columns is:

- *ent* theoretical ratio based on the entropy

[10]For $i = 0$ we have $l_0 = 1 + 0 + f(0) = 1 + B$.

[11]which is a small number since we suppose that the first values are the most frequent ones. I have used 6 in the tests.

[12]I have used 16 for that in the tests.

- *huf* Huffman coded ratio including the codetables[13]
- $\gamma\gamma$, $\delta\delta$, $\gamma\delta$ and $\delta\gamma$ the first Greek letter tells what code was used for the length encoding and the second corresponds to the code used for encoding the high bits of the distance
- *C2, C3 and C4* C2, C3 and C4 coded ratios

In *Figs. 4* and *5* one can see the ratios against the Huffman coded sizes. The ratio is computed as

$$\text{ratio} = 100\% \cdot \frac{\text{Huffman coded size}}{\text{compressed size}}$$

so the greater the number, the better the coding efficiency. In some cases the new codes outperform the Huffman codes (where the ratios are over 100%).[14]

filename	ent	huf	$\gamma\gamma$	$\delta\delta$	$\gamma\delta$	$\delta\gamma$	c2	c3	c4
bib	32.25	32.64	37.27	36.86	36.23	37.90	32.99	32.84	32.82
book1	41.63	41.83	51.57	49.77	49.09	52.26	43.05	43.05	42.97
book2	33.92	34.12	40.52	39.57	38.89	41.19	34.95	34.92	34.87
geo	69.69	70.21	81.39	82.04	79.37	84.06	73.70	73.49	73.25
news	36.92	37.23	43.06	42.01	41.31	43.77	38.32	38.14	37.99
obj1	48.28	48.96	49.51	50.43	49.69	50.26	49.42	49.22	49.04
obj2	33.17	33.54	35.85	35.78	35.18	36.46	34.28	34.33	33.99
paper1	36.57	36.96	40.80	41.01	40.26	41.55	37.43	37.32	37.31
paper2	37.97	38.37	43.91	43.77	42.98	44.70	38.96	38.77	38.77
paper3	41.24	41.69	46.67	46.86	45.95	47.57	42.21	42.06	42.06
paper4	44.46	45.09	47.42	48.79	47.74	48.48	45.34	45.21	45.14
paper5	44.23	44.93	46.58	48.04	47.01	47.60	45.08	44.92	44.88
paper6	36.80	37.29	40.17	40.73	39.93	40.96	37.63	37.57	37.50
pic	11.66	11.87	13.32	13.04	12.90	13.46	12.29	12.38	12.22
progc	35.41	35.86	38.44	38.94	38.20	39.18	36.30	36.26	36.13
progl	23.76	24.18	26.06	26.29	25.86	26.49	24.59	24.43	24.30
progp	23.51	24.01	25.39	25.77	25.28	25.89	24.35	24.19	24.09
trans	20.90	21.34	22.89	22.88	22.56	23.20	21.59	21.54	21.46
total	33.31	33.60	39.16	38.43	37.79	39.81	34.48	34.44	34.32

Fig. 2 Compression ratios of *Calgary Corpus* [%]

Finally the times needed for the compression are presented in *Figs. 6* and *7*. The values are in seconds and were measured with the unix *times()* function. The letters *O*, *C* and *D* mean *optimization*, *coding* and *decoding*, respectively.[15] As it

[13]Canonical Huffman codes were used and the codetables were stored compressed similarly as in *gzip*; see [3] for details.

[14]This is possible only because the codetable is included in the Huffman ratio. Anyway, real-life applications require the codetable too, so this is a fair comparison; nevertheless, the code parameters are also included into the ratios for the new codes.

[15]For Huffman codes *optimization* should be read as *code construction*.

filename	ent	huf	$\gamma\gamma$	$\delta\delta$	$\gamma\delta$	$\delta\gamma$	c2	c3	c4
alice29.txt	37.09	37.43	43.93	43.30	42.58	44.65	38.04	37.97	37.95
asyoulik.txt	41.09	41.52	48.59	47.85	47.12	49.32	42.46	42.35	42.33
cp.html	34.94	35.50	37.32	37.86	37.19	38.00	35.79	35.60	35.57
fields.c	29.08	29.79	30.57	31.50	30.92	31.15	29.73	29.57	29.56
grammar.lsp	35.13	36.26	36.25	37.50	36.78	36.97	35.98	35.69	35.64
kennedy.xls	19.61	20.27	31.77	32.41	31.48	32.71	28.02	28.02	28.01
lcet10.txt	34.21	34.47	41.11	40.15	39.48	41.78	35.17	35.11	35.07
plrabn12.txt	41.92	42.21	51.86	50.15	49.53	52.48	43.46	43.42	43.37
ptt5	11.66	11.87	13.32	13.04	12.90	13.46	12.29	12.38	12.22
sum	34.26	34.73	36.07	36.78	36.03	36.82	35.38	35.41	35.10
xargs.1	43.80	44.83	45.06	46.84	45.87	46.03	44.69	44.23	44.23
total	26.53	26.95	34.80	34.50	33.83	35.47	30.28	30.26	30.21

Fig. 3 Compression ratios of *Canterbury Corpus* [%]

filename	$\gamma\gamma$	$\delta\delta$	$\gamma\delta$	$\delta\gamma$	c2	c3	c4
bib	87.57	88.54	90.09	86.10	98.94	99.38	99.44
book1	81.11	84.04	85.21	80.04	97.16	97.16	97.34
book2	84.21	86.23	87.73	82.83	97.63	97.72	97.84
geo	86.27	85.58	88.46	83.53	95.27	95.54	95.86
news	86.45	88.61	90.12	85.07	97.15	97.60	98.00
obj1	98.87	97.08	98.53	97.41	99.05	99.47	99.83
obj2	93.56	93.75	95.35	92.02	97.84	97.70	98.68
paper1	90.60	90.13	91.81	88.96	98.76	99.06	99.09
paper2	87.39	87.66	89.28	85.83	98.48	98.97	98.97
paper3	89.32	88.96	90.71	87.62	98.76	99.11	99.11
paper4	95.07	92.40	94.44	93.01	99.44	99.73	99.87
paper5	96.46	93.54	95.57	94.39	99.68	100.02	100.11
paper6	92.84	91.56	93.38	91.04	99.10	99.25	99.44
pic	89.15	91.04	92.02	88.23	96.59	95.92	97.18
progc	93.31	92.09	93.90	91.53	98.80	98.92	99.26
progl	92.78	91.97	93.50	91.27	98.35	98.99	99.49
progp	94.53	93.15	94.97	92.73	98.59	99.23	99.65
trans	93.23	93.27	94.57	91.96	98.83	99.08	99.43
total	85.80	87.41	88.91	84.40	97.44	97.56	97.89

Fig. 4 Ratios to Huffman coded size of *Calgary Corpus* [%]

turns out, the new codes perform well at decompression. I must admit (again) that no special optimization was made in the coders, but this is hopefully a fair ground for comparison between them.

The totals for the *Canterbury Corpus* are somewhat weaker than I expected.

filename	$\gamma\gamma$	$\delta\delta$	$\gamma\delta$	$\delta\gamma$	c2	c3	c4
alice29.txt	85.21	86.45	87.92	83.83	98.39	98.60	98.63
asyoulik.txt	85.44	86.77	88.11	84.19	97.79	98.03	98.09
cp.html	95.12	93.77	95.47	93.43	99.20	99.72	99.81
fields.c	97.44	94.58	96.33	95.64	100.21	100.74	100.79
grammar.lsp	100.03	96.71	98.59	98.09	100.77	101.62	101.74
kennedy.xls	63.78	62.52	64.38	61.96	72.32	72.32	72.34
lcet10.txt	83.86	85.86	87.31	82.52	98.02	98.20	98.29
plrabn12.txt	81.40	84.17	85.22	80.43	97.12	97.22	97.31
ptt5	89.15	91.04	92.02	88.23	96.59	95.92	97.18
sum	96.28	94.42	96.39	94.31	98.17	98.08	98.95
xargs.1	99.48	95.72	97.74	97.39	100.31	101.36	101.36
total	77.46	78.13	79.67	76.00	89.02	89.06	89.21

Fig. 5 Ratios to Huffman coded size of *Canterbury Corpus* [%]

As it turned out, the file *kennedy.xls*[16] was responsible for this. It has extremely odd distribution for both the lengths and the distances, thus the new code performs poorly.

4.5. Some Probability Distributions

Here I present some probability distributions of a few files from the two corpora. The figures differ mainly in how well the distribution follows (or omits) the supposed exponential decay. The x-axis should be labeled as the length or distance *value* while the y-axis should read *probability*.

Figs. 8 to *11* show the length distributions of the files *obj1*, *sum*, *alice29.txt* and *kennedy.xls*, respectively. The distribution gets stranger and stranger: *obj1* has a nice decay while *kennedy.xls* is very-very extreme: just a few length values are present, the maximal probability is well over 0.5 and the corresponding length is 12!

Figs. 12 to *15* are the distance distributions of the files *plrabn12.txt*, *paper4*, *sum* and *kennedy.xls*. The same is true here as with the lengths : the distribution gets more and more extreme. Again, *kennedy.xls* is most deviant with only a few values present, with high probabilities and big gaps between the values.

[16] which is over 1M and takes more than the third of the whole corpus in size.

filename	parse	Huffman			$\gamma\gamma$		$\delta\delta$		$\gamma\delta$		$\delta\gamma$		c2			c3			c4		
		O	C	D	C	D	C	D	C	D	C	D	O	C	D	O	C	D	O	C	D
bib	1.37	0.01	0.05	0.09	0.12	0.09	0.19	0.08	0.15	0.08	0.16	0.09	0.02	0.09	0.08	0.67	0.10	0.09	12.40	0.13	0.09
book1	23.36	0.01	0.58	0.82	1.16	0.78	1.75	0.78	1.44	0.75	1.46	0.80	0.06	0.89	0.68	1.70	0.93	0.75	32.20	1.20	0.77
book2	13.67	0.02	0.37	0.53	0.73	0.51	1.13	0.51	0.92	0.49	0.94	0.53	0.06	0.59	0.45	1.75	0.60	0.51	32.72	0.78	0.52
geo	2.25	0.00	0.13	0.16	0.26	0.15	0.41	0.16	0.34	0.15	0.34	0.16	0.02	0.20	0.13	0.64	0.21	0.16	11.83	0.27	0.17
news	7.49	0.01	0.25	0.35	0.49	0.32	0.74	0.32	0.61	0.31	0.63	0.33	0.05	0.40	0.29	1.72	0.40	0.32	32.20	0.51	0.33
obj1	0.11	0.00	0.02	0.02	0.03	0.02	0.05	0.02	0.04	0.02	0.04	0.02	0.00	0.03	0.02	0.14	0.03	0.02	2.55	0.03	0.02
obj2	2.55	0.01	0.14	0.19	0.27	0.17	0.41	0.18	0.34	0.17	0.34	0.18	0.04	0.22	0.17	1.28	0.23	0.19	24.10	0.29	0.19
paper1	0.52	0.00	0.04	0.05	0.07	0.04	0.11	0.05	0.09	0.05	0.09	0.04	0.01	0.05	0.04	0.35	0.05	0.04	6.22	0.08	0.05
paper2	1.28	0.00	0.06	0.08	0.12	0.08	0.18	0.08	0.15	0.07	0.15	0.07	0.02	0.09	0.06	0.46	0.09	0.08	8.62	0.12	0.08
paper3	0.56	0.00	0.04	0.05	0.08	0.04	0.11	0.04	0.09	0.04	0.09	0.05	0.01	0.05	0.04	0.30	0.06	0.04	5.53	0.08	0.05
paper4	0.08	0.00	0.01	0.01	0.03	0.01	0.04	0.01	0.03	0.02	0.03	0.01	0.01	0.01	0.01	0.11	0.02	0.01	1.99	0.03	0.01
paper5	0.06	0.00	0.01	0.01	0.02	0.01	0.03	0.01	0.03	0.01	0.03	0.01	0.01	0.01	0.01	0.09	0.02	0.01	1.75	0.02	0.01
paper6	0.31	0.00	0.02	0.03	0.06	0.03	0.08	0.04	0.07	0.03	0.07	0.03	0.01	0.04	0.03	0.25	0.05	0.03	4.69	0.05	0.03
pic	4.00	0.06	0.12	0.20	0.20	0.18	0.30	0.17	0.25	0.17	0.26	0.18	0.06	0.16	0.16	1.69	0.17	0.18	34.28	0.21	0.19
progc	0.30	0.00	0.02	0.03	0.06	0.03	0.08	0.04	0.07	0.03	0.07	0.03	0.00	0.05	0.03	0.24	0.05	0.03	4.47	0.06	0.04
progl	0.65	0.00	0.04	0.05	0.06	0.04	0.10	0.05	0.08	0.05	0.08	0.05	0.02	0.05	0.04	0.41	0.05	0.04	7.70	0.07	0.05
progp	0.32	0.01	0.02	0.03	0.04	0.03	0.07	0.03	0.05	0.03	0.05	0.03	0.01	0.03	0.02	0.35	0.04	0.03	6.64	0.05	0.03
trans	0.60	0.00	0.04	0.06	0.07	0.04	0.11	0.05	0.09	0.05	0.08	0.05	0.02	0.06	0.05	0.57	0.05	0.05	10.58	0.07	0.05
total	59.48	0.13	1.96	2.76	3.87	2.57	5.89	2.62	4.84	2.52	4.91	2.66	0.43	3.02	2.31	12.72	3.15	2.58	240.47	4.05	2.68

Fig. 6 Calgary Corpus times

filename	parse	Huffman			$\gamma\gamma$		$\delta\delta$		$\gamma\delta$		$\delta\gamma$		c2			c3			c4		
		O	C	D	C	D	C	D	C	D	C	D	O	C	D	O	C	D	O	C	D
alice29.txt	2.92	0.01	0.10	0.14	0.21	0.13	0.32	0.14	0.26	0.14	0.26	0.14	0.02	0.16	0.12	0.83	0.16	0.13	15.40	0.21	0.14
asyoulik.txt	2.48	0.00	0.10	0.13	0.19	0.12	0.29	0.12	0.24	0.12	0.24	0.12	0.03	0.14	0.11	0.70	0.15	0.12	13.20	0.19	0.12
cp.html	0.12	0.00	0.02	0.02	0.03	0.02	0.05	0.02	0.04	0.02	0.04	0.02	0.00	0.03	0.02	0.18	0.03	0.02	3.46	0.03	0.02
fields.c	0.05	0.00	0.01	0.01	0.01	0.01	0.02	0.01	0.02	0.00	0.01	0.01	0.01	0.01	0.00	0.09	0.01	0.00	1.78	0.01	0.00
grammar.lsp	0.03	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.01	0.04	0.01	0.01	0.81	0.00	0.01
kennedy.xls	20.57	0.01	0.33	0.42	0.86	0.59	1.29	0.62	1.07	0.58	1.08	0.61	0.01	0.67	0.54	0.20	0.67	0.59	3.77	0.87	0.63
lcet10.txt	9.58	0.01	0.27	0.38	0.52	0.36	0.79	0.38	0.66	0.35	0.67	0.37	0.06	0.41	0.32	1.66	0.43	0.35	31.29	0.55	0.36
plrabn12.txt	14.17	0.01	0.36	0.50	0.73	0.48	1.09	0.50	0.91	0.47	0.92	0.51	0.06	0.56	0.43	1.67	0.59	0.47	31.35	0.76	0.48
ptt5	4.01	0.06	0.11	0.20	0.20	0.18	0.31	0.18	0.25	0.18	0.26	0.18	0.07	0.16	0.17	1.69	0.17	0.19	34.24	0.22	0.18
sum	0.27	0.00	0.02	0.03	0.04	0.03	0.07	0.03	0.06	0.03	0.06	0.03	0.01	0.03	0.02	0.21	0.04	0.03	3.99	0.05	0.03
xargs.1	0.02	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.01	0.00	0.01	0.01	0.00	0.01	0.00	0.05	0.01	0.00	0.88	0.01	0.01
total	54.22	0.10	1.33	1.84	2.80	1.92	4.25	2.01	3.53	1.89	3.55	2.01	0.27	2.18	1.74	7.32	2.27	1.91	140.17	2.90	1.98

Fig. 7 Canterbury Corpus times

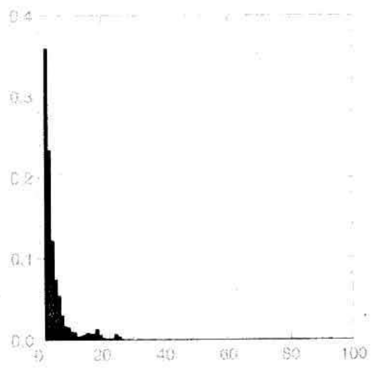

Fig. 8 obj1 lengths

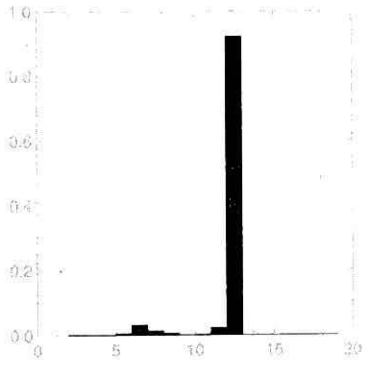

Fig. 11 keneddy.xls lengths

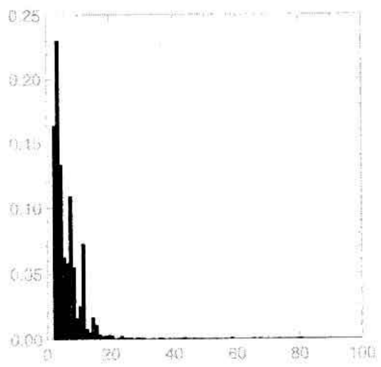

Fig. 9 sum lengths

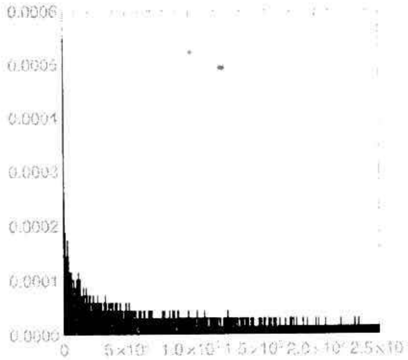

Fig. 12 plrabn12.txt distances

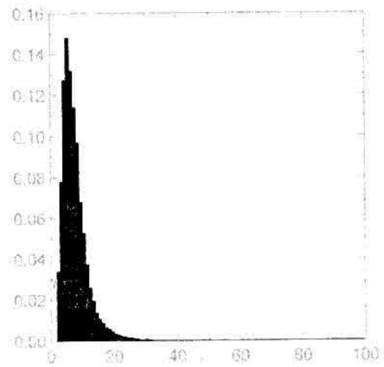

Fig. 10 alice29.txt lengths

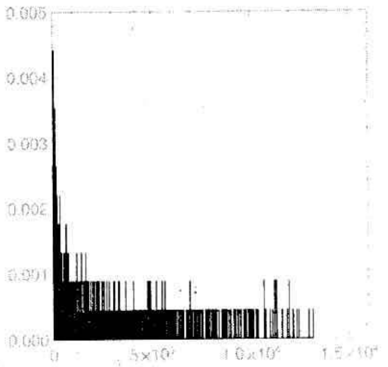

Fig. 13 paper4 distances

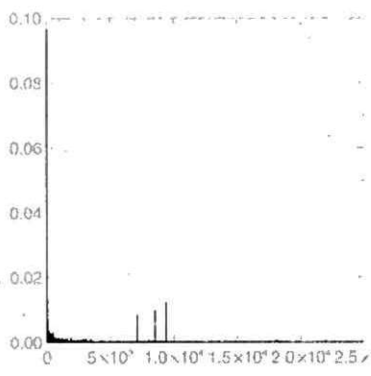

Fig. 14 sum distances

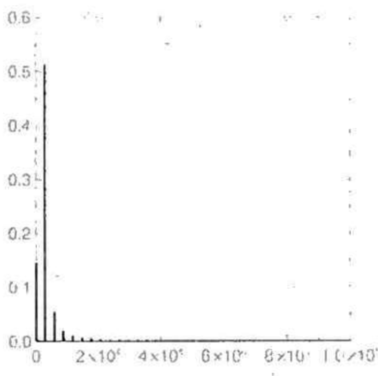

Fig. 15 kennedy.xls distances

5. Possible Future Work

Another possible application of the codes could be the index compression for full-text databases. The gaps between the indices seem to have a similar decaying distribution as discussed here. The ones interested may find an excellent discussion of index compression in [9].

It might be worth trying to restrict the parameters P and Q to be integer powers of two. In that case the multiplication and division can be performed by bit shift operations, thus the codes could be used effectively on even 8 bit microprocessors/controllers.[17]

A *stop* parameter could be introduced similarly as with the start-step-stop code. This would save one bit from the unary code for the largest encoded values, but since these values are the most rare ones, the savings might be just marginal.

As seen from the sample distributions, there are cases where the exponential decay holds just partly or not at all. For these a different kind of $f(u)$ should be chosen.

6. Conclusions

I have presented a new family of codes that are parametrized by a few integers. The codes performed quite well on the test files, nearly as good as Huffman coding. The advantages of these codes are fast decoding and little memory requirements, the disadvantage is that the parameters have to be optimized and this increases the time needed for compression.

[17]Of course, this will sacrifice some of the efficiency of the code.

References

[1] ZIV, J. – LEMPEL, A., A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory*, **23** (1977).

[2] STORER, J. A. – SZYMANSKI, T. G., Data Compression via Textual Substitution, *Journal of the ACM*, **29** (1982).

[3] DEUTSCH, P., DEFLATE Compressed Data Format Specification Version 1.3, *Network Working Group, Request for Comments: 1951*, 1996.

[4] FIALA, E. R. – GREENE, D. H., Data Compression with Finite Windows, *Communications of the ACM*, **32** (1989).

[5] KATAJAINEN, J. RAITA, T., An Analysis of the Longest Match and the Greedy Heuristics in Text Encoding, *Journal of the ACM*, **39** (1992).

[6] WITTEN, I. H. – BELL, T. C., The Relationship between Greedy Parsing and Symbolwise Text Compression, *Journal of the ACM*, **41** (1994).

[7] SHANNON, C. E., A Mathematical Theory of Communication, *Bell Systems Technical Journal*, **27** (1948).

[8] ELIAS, P., Universal Codeword Sets and Representations of the Integers, *IEEE Transactions on Information Theory*, **21** (1975).

[9] WITTEN, I. H. – MOFFAT, A. – BELL, T. C., *Managing Gigabytes*, Van Nostrand Reinhold, New York, 1994.

[10] GOLOMB, S. W., Run-length Encodings, *IEEE Transactions on Information Theory*, **12** (1966).