Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di laurea specialistica in Informatica

Tesi di laurea

On the ACB compressor

Laureando:
Jonatan Masci

Relatore:
Prof. Paolo Ferragina

Controrelatore:
Prof. Giorgio Gallo

Anno Accademico 2008/2009

Abstract

Context-based compression methods are the most powerful approaches to squeeze arbitrary textual data. They offer a good predictive model for the subsequent data based on the already seen one, without assuming any probability distribution for the input source. In this thesis we analyze the adaptive ACB method [8] which is mostly unexplored in the literature, although preliminary results showed compression ratios comparable (or even superior) to the best known data compression utilities.

The novel feature of ACB consists of deploying both the previous context and the subsequent content to find a succinct encoding for the latter one. We perform a large set of experiments to study the experimental behavior of ACB and to compare it with known compressors, thus devising variations of the basic ACB-scheme that result promising for future developments.

Acknowledgments

First of all I would like to thank my advisor, Prof. Paolo Ferragina, for supporting me over this work and for giving me the opportunity to work with him on this project.

Thanks to my friends, in particular my Casapiddu's flatmates The Weird Boy, Rambo, El Mangi and La Sguattera who have become my second family in these years I lived in Pisa. A special thanks goes to my old friends, Daniele and Andrea ('i pedri'). Thanks also to Mahroo without whose this work would have had less importance.

Last but not least I have to thank my family who has always trusted my decisions and believed in me also when I have had doubts about my future.

Contents

Chapter 1

Introduction

Recent years have been characterized by an exponential increment of stored information: gathered from the web, from our credit card logs or just when we send an sms or turn-on our mobile phone, just to cite a few. By now, everything we do is logged into a file, the big bro is not so far and in this case there is no someone who watches us, we want us to be watched.

The web, as always, has a dominant place also into this issue, and the social networks' boom confirms that. Everything is stored and everything is considered important. No one cares if that particular information will never be used again along the whole universe life, the important thing is to have such information. These are also the years where big companies buy other companies spending millions of dollars just to take their query logs to do data mining on that. This confirms again that information is power. Data mining is a relatively recent discipline who has been developed in concomitance with this expansion in order to understand the relations and the associations that humans cannot see due to the enormous amount of data: *We are living in the information era.*

All of this amount of data can be separated into two parts, the first one representing the information that has to be always available and the second one representing the information that could be stored into archives and retrieved when needed.

In the former case we have, for example, the web indexes where we cannot pay too much to retrieve them during query processing. Here we have to work in real time[20][11].

In the latter case we have all the data that are not used on the fly or very often such as query logs, backups and cached pages of search engines for example. In these cases we can pay a little overhead to retrieve the information since this is justified by the lower space usage (and bandwidth usage if we are on the web) and unfrequent access.

Since we are speaking about millions of gigabytes we have to take into account also the amount of money needed to store that data. It is true that mass storage devices' prices have decreased but it is also true that the amount of information to store has grown in a measure that has kept the "cost-per-unit-of-storage" relation nearly unvaried (ratio between the need of space and the cost of the space needed).

In this thesis we are interested in *lossless* compression methods to apply to the latter case presented. These techniques are distinguished from the more general *data* compression methods because the original file can always be reconstructed exactly. This is why they are also known as *text* compression methods. In fact, for some types of data other than text such as sound or images, small changes, or *noise*, in the reconstructed data can be tolerated because it is a digital approximation to an analog waveform anyway.

Lot of techniques have been invented and reinvented over the years, departing from one of the earliest and best-known methods, Huffman coding, and arriving to the best and most used compressors which can be considered the state-of-the-art: gzip [6], bzip2 [3] and PPM [9].

The first one, *gzip*, is an adaptive dictionary based method. It is a variant of the Lempel-Ziv schemes developed in the 1970s [28], the LZ77 and LZ78. The idea behind these methods is the following: a substring of text is replaced with a pointer to where it has occurred previously. The many variants of Lempel-Ziv coding differ primarily in how pointers are represented.

In Figure 1.1 we present an example of the LZ77 compression where the characters `abaabab` have already been decoded, and the next characters to be decoded are represented by the triple `<5,3,b>`. The decoder will go back five characters and will copy three characters, yielding the phrase `aab`. Then it will add the third item, the `b` character. The Lempel-Ziv methods, which are theoretically optimal, in practice do not reach the compression ratio of the other compressors. On the other hand they are relatively easy to implement, requires a small amount of memory and decodes the text extremely quickly.

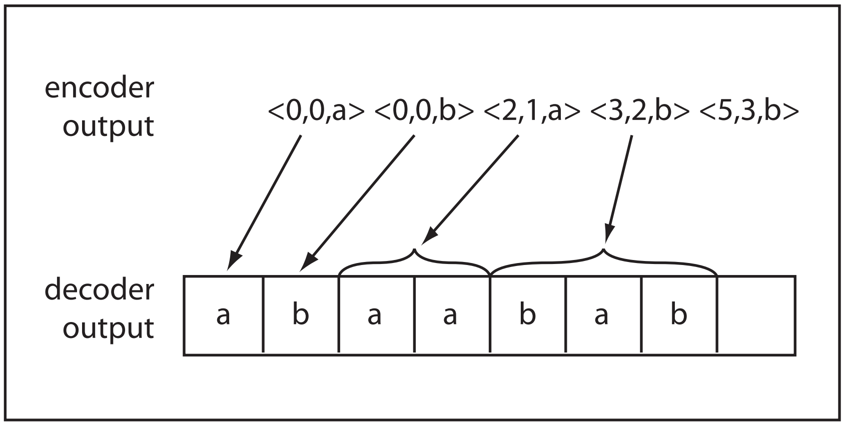

Figure 1.1: Example of LZ77 compression

Bzip2 is a block-sorting method based on the Burrows-Wheeler Transform (BWT [24]), first published in 1994 and deeply studied in the literature. The BWT permutes the characters in a text so that those occurring in similar contexts end up near each other, producing a more compressible data. The transformation is performed by sorting each character in the text, using its context as the sort key. In Figure 1 there is an example of the BWT. First we create a matrix with all the permutations of the text, then we sort all the suffixes and then we take the last column as result. This transformation is easily reversible thus the order in which corresponding characters appear in the two columns (the first and the second of the permuted text) are the same. The method used to code the permuted string is crucial and since the characteristics of the normal text are not preserved, a context-based coder is not appropriate. One suitable method, which is also the one used in bzip2, is to use a move-to-front coder (MTF), which assigns higher probability to characters that have occurred recently in the input. For the final step it uses the Huffman coding because of the patent issue about the arithmetic coder which was used in bzip. The pros of bzip2 is that it achieves the state-of-the-art compression ratio at a reasonable fast speed.

1	`mississippi#`
2	`ississippi#m`
3	`ssissippi#mi`
4	`sippimis#mis`
5	`issippi#miss`
6	`ssippi#missi`
7	`sippi#missis`
8	`ippi#mississ`
9	`ppi#mississi`
10	`pi#mississip`
11	`i#mississipp`
12	`#mississippi`

(a)

1	`#`	`mississipp`	`i`
2	`i`	`#mississip`	`p`
3	`i`	`ppi#missis`	`s`
4	`i`	`ssippi#mis`	`s`
5	`i`	`ssissippi#`	`m`
6	`m`	`ississippi`	`#`
7	`p`	`i#mississi`	`p`
8	`p`	`pi#mississ`	`i`
9	`s`	`ippi#missi`	`s`
10	`s`	`issippi#mi`	`s`
11	`s`	`sippi#miss`	`s`
12	`s`	`sissippi#m`	`i`

(b)

1	`i`
2	`p`
3	`s`
4	`s`
5	`m`
6	`#`
7	`p`
8	`i`
9	`s`
10	`s`
11	`s`
12	`i`

(c)

Figure 1.2: BWT of the string `mississippi#`: (a) rotations of the string, (b) sorted by contexts, (c) permuted string

The prediction by partial matching (*PPM*) technique was firstly introduced by Cleary and Witten in the 1984 and uses the already seen contexts to build a statistical model to predict next symbols of the input stream. It uses the conditional probabilities, so for example if seen `th` 12 times followed by `e` 7 times, then $P(e|th) = 7/12$, the probability of to have an `e` for the context `th` is 7/12. Much of the proposed PPM versions differ in how they handle inputs that have not already seen. Since we cannot code 0 probabilities, the obvious solution is to code a *never-seen* symbol which triggers the escape sequence [19]. This implementation starts with a context of fixed length that is shortened until the prediction can be made. The main drawback of this compressor is its high resources requirements while its pro is the effectiveness which reaches or exceed the bzip2's one.

Among the best compressors briefly reported here we asked ourself for the existence of an hybrid method who should merge the pros of the dictionary based model, the intuitiveness and the simpleness, and the pro of symbolwise models, the effectiveness.

For that reason we investigated an hybrid method who takes the idea of context sorting from bzip2 and PPM and that tries to mix it with the really simple and intuitive concept of the Lempel-Ziv schemes, without assuming any probability distribution for the input source.

In the literature such method has been proposed by George Buyanovsky in the 1994 and is called ACB [8]. After the first work, which is in russian, there have not been other works investigating this approach.

In this thesis we analyzed this adaptive method which is mostly unexplored in the literature, although preliminary results showed compression ratios comparable (or even superior) to the best known data compression utilities.

The novel feature of ACB consists of deploying both the previous context and the subsequent content to find a succinct encoding for the latter one. We perform a large set of experiments to study the experimental behavior of ACB and to compare it with known compressors, thus devising variations of the basic ACB-scheme that result promising for future developments.

1.1 Organization of the Thesis

The rest of this thesis is organized as follows.

Chapter 2 provides an overview of the basic concepts of this thesis.

Chapter 3 describes the implementation of the method and the related issues.

Chapter 4 presents the whole corpus of experiments and the results obtained under the space occupancy, memory usage and compression ratio aspects.

Chapter 5 illustrates some methods which should actually be used in order to improve the ACB's performance. We present briefly also a variation of ACB used to do some brute force test.

Chapter 6 concludes.

Chapter 2

Context-Based Data Compression

once upon a ...

time, of course. Everyone, or better, most of people, will give this continuation to the previous sentence, simply because the word *time* is the most suitable word for the given text.

Some of the most powerful compression methods exploits the nature of the input, such as in the example before. These methods, instead of to treat all data with the same rules, are like a sort of specialization built over the specific instance of the problem. One of the most interesting portion of them is the one based on the Shannon's classic paper on information content of English text where he established the well-known bounds of 0.6-1.3 bits per letter [23].

2.1 Shannon on English text compressibility

Shannon in his famous work on English text [23] describes two methods. In both of them a person is asked to predict letters of a passage of English text where some of the preceding text may be made available, but where the text to be predicted must be unfamiliar to the subject. Shannon also shows that the *responses* to the predictions are equivalent to the original text and that an "identical twin" (or its mathematical equivalent) could be used to recover the original input. In both cases the person effectively prepares a ranked list of the probable symbols, most probable first, and present the list to the comparator:

- **first method**. The person predicts the letter and is then told "correct", or is told the correct answer.
- **second method**. The person must continue predicting until the correct answer is obtained. The output is effectively the position of the symbol in the list and the sequence of "NO" and the final "YES" responses is a unary-coded representation of that rank or position.

A third method is a hybrid of the two given by Shannon and it is explained in [10]. After some small numbers of failures (typically 4-6) the response is the correct answer, rather than "NO". With some types of coding for the prediction values this may give a more compact code.

guesses	1	2	3	4	5	>5
success rate	71%	10%	7%	2%	2%	8%

Table 2.1: Shannon's original prediction statistics.

The distribution is very highly skewed and it means that we have a low information content per symbol which implies great compressibility.

Shannon used a technique that fall under the name of "symbol-ranking". In fact, usually, compressors are "symbol-frequency" based, such as Huffman, and they process input in order to assign shorter codeword to more frequent symbols. The "symbol-ranking" method on the contrary starts from the current context, what it is known, and prepares a list of symbols that most likely will follow the context.

There exists several implementations based upon this idea, the most interesting ones for our intents are $(\alpha, \beta) - HYZ$, described in [18] [14] and ACB that we are going to investigate in our thesis. An implementation for the first one has been proposed by Yokoo in [26].

2.2 Notation

Given a string T of length n we define with $T[i]$ the character at i-th position in T $(i < n)$ and with $T[i : j]$ the substring (text between two included indices).
For example if $T =$ `CASAPIDDU` we will have $T[0 : 2] =$ `CAS`. A substring can be also reversed if $i \geq j$, so $T[2 : 0] =$ `SAC`.

We introduce now two fundamental concepts to develop context-based compressors: context and content. A **context** is a substring already seen read right-to-left. Some contexts for T are `SAC`, `PASAC` and `C`. Formally speaking we define it using the following formula

$$\mathit{context}(i) = T[j : i] \text{ such that } i < j, j < n, i < n \tag{2.1}$$

For a context we also define its **order** by introducing the number of symbols we are considering for each context. Two order-three contexts for T are `SAC` and `DDI`.
A **content** is the portion of text that follows a context. For example for the context `SAC` the corresponding content is `APIDDU`. The formal definition is the following

$$\mathit{content}(i) = T[i + 1, n - 1] \tag{2.2}$$

As we have done for the context we also define the **order** of a content in the same way.

So we have as many contexts and contents as how many characters our text is composed by. Assuming that we have read a string until the i-th position, we define the **look-ahead** buffer as $\mathit{content}(i)$.

2.3 $(\alpha, \beta) - HYZ$

Formally speaking the $(\alpha, \beta) - HYZ$ compression method works by replacing disjoint blocks (substrings) of size β with shorter codewords. The codeword selection is done as follows. Say the first $i - 1$ blocks have been encoded. To compute the codeword c_i for block i we determine its context first. The context of the i-th block $T[i\beta : (i + 1)\beta - 1]$ is the longest substring $T[k : i\beta - 1]$, of size at most α such that $T[k : (i + 1)\beta]$ occurs earlier in T. The codeword is the ordered pair $< \gamma, \lambda >$ where γ is the length of the context of block i and λ is the rank of block i with respect to the context in accordance to some predetermined ordering. For instance, we can use the lexicographic ordering of all distinct substrings of size β that follow any previous occurrence of the context of block i in T.

In Figure 2.2 there is an example where the β-blocks following some context are ordered by their position in T. The output for the reported situation is `<2, 0>`. Note that if we had not have had the `bb` context we would have restricted the research to the `b` case.

```
T = bbababaababbbabb | abab
```
$\alpha = 2$, $\beta = 4$

context of block `abab` = `bb`

list of earlier occurrences of this context:

```
bb | abab
bb | babb
bb | abb
```

Table 2.2: $(\alpha, \beta) - HYZ$ example.

2.3.1 Yokoo's implementation

One $(\alpha, \beta) - HYZ$ implementation has been proposed by H. Yokoo in [26] and it considers the case when $\beta = 1$ and α is unbounded. Its implementation encodes one symbol at a time and the encoding process consists of several operations faced to compute the symbol ranking. Given a string it sorts all the previous contexts by the length of their common suffixes. Then, it enumerates all the candidates for the symbol to encode in the order of the next symbol of each context. The rank is represented by the count of distinct symbols, the bracketed value in the example. If no match is found, then it encodes a virtual out-of-bound rank followed by the look-ahead character. In Table 2.3 there is an example for $T =$ `bacacaba`, the λ identifies the empty string and x identifies the look-ahead character. If $x =$ `c` we will emit a rank 0 thus the first `c` we met is on the group who shares most symbols and if $x =$ `a` we will emit a rank 2 since the `a` is on the 0-similarity group.

contexts	
λ	b
b	a
ba	c
bac	a
baca	c
bacac	a
bacaca	b
bacacab	a
bacacaba	x

similarity	contexts	symbol rank
	bacacaba	x
2	ba	c (0)
1	baca	c
	bacaca	b (1)
0	λ	b
	b	a (2)
	bac	a
	bacac	a
	bacacab	a

Table 2.3: Yokoo's example: on the left the previously seen contexts with the first following character, on the right the same contexts sorted by similarity group with T

The context sorting is done using a fixed-order context (maximum number of character to compare) so, in order to measure the similarity between two contexts, it introduces a parameter M. The relation for a context corresponds to the lexicographic order of at most M symbols, so for any integer $M > 0$ we have

$$x[1..m] \overset{M}{\prec} y[1..n] \tag{2.3}$$

if one of the following two conditions holds:

1. we have $x[m-i] \prec y[n-i]$ for an integer i such that $0 \leq i < min\{m, n, M\}$ and $x[m-j] = y[n-j]$ for an integer j such that $0 \leq j < i$.

2. we have $m < n, m < M$ and $x[m-i] = y[n-i]$ for any integer i such that $0 \leq i < m$.

The encoding process can be schematized as following:

- find the position p of the substring to encode in the context dictionary
- merge the list of 4 contexts in the neighborhood of p (2 up and 2 down). It is not mentioned how to do this and why 4 contexts.
- get the rank of the symbol $x[i]$ that we wanted to encode counting the number of distinct symbols prior $x[i]$ in this sorted sequence. If no symbol will be found the out-of-bound rank will be emitted
- first symbol is transmitted as it is

Yokoo proves that the average length of a codeword representing a β-sized block ($\beta = 1$) approaches the conditional entropy for the block, $H(C)$, within an additive term of $c_1 \log H(C) + c_2$ for constants c_1 and c_2, provided that the input is generated by a finite-order Markovian source. However we did not find any clarification about how big are these constants. The important thing is that Yokoo proved also that by encoding a block of d symbols at a time, it is possible to achieve asymptotically the entropy bound as closely as desired.

2.4 ACB

The ACB compression method is a method proposed by George Buyanovky, the name stands for "Associative Coder of Buyanovsky", of which not so many details are known. The only documentation available is outdated and in Russian [8]. An English informal interpretation is available at [1] [2] and that is what we have taken as a starting point of our work.

To explain the key principles of the algorithm we begin with an example. Assume that the text `‘‘swiss miss is missing...’’` is part of the input stream and that the first seven symbols have already been encoded and that are now in the dictionary of the previously seen contexts, sorted in accordance to the reverse lexicographic ordering as shown in Table 2.4. The look-ahead buffer starts with the string `‘‘iss is...’’`. We want to replace a prefix of this string with a shorter codeword exploiting context-content relations.

`swiss m|iss is missing...` $\leftarrow$ text to be read

For each element we also have the text following it, the content string. Now imagine to act as LZ77, we should search for the longest common substring in the previously seen text for the look-ahead buffer, that is at position 2. In ACB we search it in the content part of our dictionary (Table 2.4), item 7, and then we encode it as difference from the best context matched departing from obvious one.

In practice it tries to exploit the Shannon result in order to copy phrases and not single symbols. To do that it chooses the most similar context to the obvious one as the sentence-predictor. We describe now the encoding and decoding process in details.

2.4.1 Encoding

Continuing with the example we gave before, the current context `‘‘swiss m’’` is matched to the dictionary entries. The best match, right-to-left, have to be chosen between items at position 2 and 4. The encoder will choose the

Unordered		Ordered
`s|wiss m`	1	`swiss |m`
`sw|iss m`	2	`swi|ss m`
`swi|ss m`	3	`swiss m|... <- obvious context`
`swis|s m`	4	`s|wiss m`
`swiss| m`	5	`swis|s m`
`swiss |m`	6	`swiss| m`
`swiss m|...`	7	`sw|iss m`

Table 2.4: Dictionary of the first six contexts before and after sorting.

one who shares the longest backward common prefix (`bw_lcp`) and, in case they have the same `bw_lcp`, it assumes that the encoder select next entry, 4 in our case. The content is then matched in the content part of the dictionary, best match for the look-ahead buffer text occurs at position 7 and is 4 characters length. The output is then the triplet `<7-4, 4, i>` where the first element is the distance between the best matching context index and the best matching content index within the dictionary, the second element is the length of the match (the substring to copy of the content) and the third one is the first unmatched symbol. Then, all five contexts are added to the dictionary which becomes the one shown in Table 2.5.

The new buffer is

`swiss miss i|s missing...` ← text to be read

The process iterates until the end of the buffer is reached. In case the length of the match is 0 then the triplet `<0, 0, c>` is emitted, where `c` is the first character in the look-ahead buffer.

2.4.2 Decoding

The decoder will fill first the dictionary with some startup contexts and then, for each triplet, it will emits the portion of the identified text plus the look-

`s|wiss miss i`	1	`swiss miss |i`
`sw|iss miss i`	2	`swiss |miss i`
`swi|ss miss i`	3	`swiss miss i|...`
`swis|s miss i`	4	`swiss mi|ss i`
`swiss| miss i`	5	`swi|ss miss i`
`swiss |miss i`	6	`swiss m|iss i`
`swiss m|iss i`	7	`s|wiss miss i`
`swiss mi|ss i`	8	`swiss mis|s i`
`swiss mis|s i`	9	`swis|s miss i`
`swiss miss| i`	10	`swiss miss| i`
`swiss miss |i`	11	`swiss| miss i`
`swiss miss i|...`	12	`sw|iss miss i`

Table 2.5: Dictionary after first step.

ahead character read from the input triplet. After each step it will update its dictionary. The context is determined using the same rules so that the best matching context will be the same used in compression. This is easy to prove since we never used the obvious entry while encoding.

For example, assuming we have the context dictionary shown in Table 2.4, the triplet `<3, 4, i>` that has been emitted in compression will be processed as follows:

- compute the `bw_lcp` between the obvious context and its predecessor/successor and then select the best matching one. In case they have the same `bw_lcp`, such as has been done during the encoding process we choose the next item, which is the number 4 in our case
- offset to item at position $4 + 3 = 7$
- append the 4 characters substring of content (`iss m`) to the output buffer (the decoded text)
- append the look-ahead character `i`

The resulting output, after this step, will be `swiss miss i|...`. The process iterates until the end of the triplets will be reached.

2.4.3 On comparing LZ vs ACB

Now that we have a clearer, although not complete, vision of how ACB works we can analyze the similarity existing with the Lempel-Ziv compression scheme and its variants. Both compressors, in fact, copy substrings of text accessing them by a distance, it is just how we interpret this distance that makes the difference.

In LZ77 we have a fixed size, say k, sliding-window which identifies the portion of the text where to search for the longest substring who matches the look-ahead buffer. The dictionary here is represented by the context of order k. The distance is the offset between the current position and the position of the substring who has the longest match.

In LZ78 the dictionary contains the substrings already seen, so it is a real data structure not a simple string. The distance is the index of the matched phrase within this dictionary.

Also in ACB we have distances but the semantic associated is really different. Instead of identifying the substring to copy by its position in the text or by its index we identify the distance as the offset between the index of the item representing the best substring to copy and the index of the item representing the look-ahead buffer within the dictionary built over all the suffixes ordered by the reverse lexicographic relation.

This is a very powerful approach because in LZ the distance emitted between two indexes in the text is not related to any kind of relation since we search for the content without taking the context into account in any manner. In ACB instead we try to predict the content to copy, we depart from context and not from content. The cost we pay to encode this distance is deeply related to the prediction error as we reported in Section 4.1.

Under a software engineering point of view we have the same I/O interfaces both for LZ and ACB. We have two interchangeable black-boxes which share the same triplet but who have really different cores.

2.4.4 Why we have chosen it

ACB can be considered as the most general $(\alpha, \beta) - HYZ$ implementation, with α and β unbounded and, since in literature this class of compressors has been proved to achieve the conditional entropy bound of the input source, it is very powerful in theoretical terms. What have always discouraged the research of new solutions based on the $(\alpha, \beta) - HYZ$ method is mainly the low efficiency of the implementations which seems to be unavoidable.

In fact, if we look at the Yokoo's implementations, which is an $(\alpha, 1) - HYZ$ with α fixed to a maximum of 8 characters, we note that it is not very efficient and that the overall compression rate is not awesome. We think that this is due to the limit of one character copy ($\beta = 1$) which involves a number of steps equals to the length of the input text. This means that we have to encode such number of elements, which is penalizing also for very compressible data.

In our case we have

- a different distance computation and a different dictionary,
- an unlimited phrase's length copy,
- an unlimited and not fixed context length

We think that for real inputs our dictionary organization, illustrated in Section 2.4, in concomitance with the policy chosen to get the offset for the content will produce quite small distances accompanied by long copies. The number of iterations should decrease of a factor equals to the ratio between 1 and the mean substring length of ACB. For example, if Yokoo takes n iterations, where n is the length of the text, we will use $n * m$ iterations, where $m = 1/mean\_copy$ is the β-gain.

More the prediction will be accurate and smaller will be m. What we have to understand is if to a really small β-gain factor correspond also short distances.

While we are pretty sure that the distance coding will be good, we are not so sure about the efficiency issues which we could inherit from $(\alpha, \beta) - HYZ$. This is not the main problem of this thesis but we think that this will be the main problem if we will want a production version of ACB.

Chapter 3

Implementing ACB

The development of ACB has been a tricky experience, due to the complexity of managing all the sorted suffixes of the text.

For example

- after getting the best matching context we want to search all the available contents in order to get the best matching content and we would like to do it without scanning the whole data
- we should have also the possibility to compute the distance between the two item in the dictionary avoiding another linear scan to get this offset (distance to emit).

It is clear that we need a data structure that is able to perform efficient searches and that provides a method to access elements by their index.

In compression we did not encountered lot of problems because we have all the text and so we can build a static data structure such as a suffix-array [20][11][5]. In decompression instead, we need a dynamic data structure that is able to handle millions insertions, searches and that can allow random access to items (and that preferably implements the iterator pattern).

These sophisticated data structures have a great impact on the overall efficiency of the algorithm and cannot be underestimated although our main target is to investigate the compression efficacy of this approach compared to the best known ones in literature.

The language we used to implement the compressor and the libraries it uses is mainly `C`, with just some extensions of `C++` only when needed.

During the rest of the chapter we illustrate how we transformed the theories exposed before into code.

3.1 Compression

The ACB's dictionary is based upon the reverse lexicographically context sorting so we need at least a data structure to keep them organized. The content part of each item can be easily retrieved so it is not absolutely necessary another data structure for them. Here the problem is, as already mentioned, that for a given context (the one corresponding to the current symbol read) we want first to retrieve its placement among the seen contexts in the dictionary. Then we want to get the item whose has the longest common prefix with the look-ahead buffer. We can solve it by

- taking only one data structure for the context and scanning the whole items looking at the associated content. The brute force approach.
- using another data structure to store the sorted contents. The faster but memory hog solution.

We have obviously chosen the second option because the first one is so computationally expensive that will preclude tests just after some kilobytes of data. So, in this phase, we used two suffix arrays, one for the context dictionary (SA^R) built over the reversed text T^R and one for the content dictionary (SA) built over T. This allow us to search the best content in logarithmic time and to retrieve the gap between two contexts in $O(1)$.

This choice bring us to another problem due to the static nature of the suffix array. Building the data structure all over the text involves an availability check for each item because we cannot use a content or context that we have not already seen from input. In practical terms we can not output a triplet in which we tell the decoder to copy a substring that is located after the actual position. Moreover when we compute the offset we have to ignore the unavailable items to be coherent with the decompression procedure.

In order to get the item who has the longest common prefix with the text to encode we exploit the ordering of the items so, instead of to do a search, we look at the first available item upward and downward respect to the one corresponding to the obvious content. Since they are sorted these elements are the most similar to the given one.

After we have remarked how we find the best item we can list the possible solution to the problem:

- scan all the items while going upright and downright checking the item's availability at each step.
- scan only the existing ones:
  - using a static data structure, such as a bit array, that keeps track of inserted elements and that provides iterations functionalities among only the available items.
  - using a tree containing only the available items

The first method is highly inefficient at the beginning because of the lack of available items while it is efficient at the end so in the mean case it involves at least in a number of checks which we consider too much since we have to do them both for the context and the content search. This method has also the problem of computing the offset between the context and the content in efficient time.

The third one involves a scan to compute the distance between the context and the content. This solution has not a random access to its elements so at each step we have to do a search. It has also a space consumption which is bigger than the one needed by the the second solution.

This is why we have opted for the second choice, the best trade-off. The offset will be computed in very efficient time, while the occupied space is almost optimal. We have just a little bit time overhead due to the bit packing stuff.

We called such data structure, developed for the scope, the **bit_ranker**. The features we implemented are the only needed ones:

- `insert(i)`: set true the bit for element i
- `rank(i)`: get the number of 1s before i
- `prev(i)`: get the position of the first 1 preceding i
- `next(i)`: get the position of the first 1 following i

Each element is represented by a bit, which is packed into a block of unsigned char for fast serialization feature. Blocks are organized into super-blocks which keep the process of counting the available elements contained. We could have putted the number of the following elements in each super-block instead of the number of contained elements to get a faster ranking procedure. The fact is that in our case we are interested in a fast insert function because it will be called n times, where n is the input length, while the ranking procedure will be called only at each step. Since we hope these steps to be really smaller than n we opted for a fast insert function which will be penalized introducing the sum of the following available elements.

In Figure 3.1 there is an example of how the data structure is organized.

Each element of the `bit_ranker` represents an element of the suffix array so, assuming that we are at position *pos* in T, all the bits corresponding to values of k such that $SA[k] = i$ and $i <= pos$ are set to true.

There are other methods to do it, and this is not the best one under the complexity point of view, but it is a very good compromise in terms of coding time so for the moment we have not planned any substitution. A further modification could consist in the introduction of another super-block level for example.

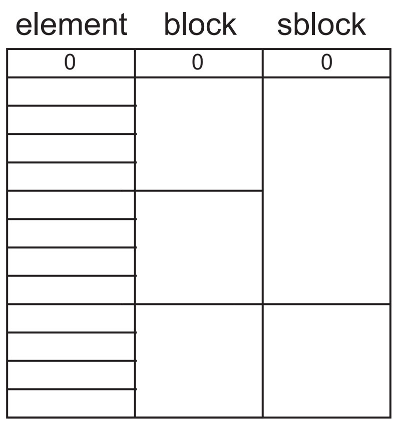

Figure 3.1: bit_ranker data structure overview

We know that we have opted for an implementing solution which has a memory overhead for the additional suffix array but, hoping distances will be as small as expected, in future we could use only one data structure. For the moment, however, we have to use a whole-range approach, because we have to prove all our assumptions and because we have to test the algorithm for optimal cases and not for sub-optimal ones.

With the implementation given, for each character i in the input stream, we

- compute best context for $T[i:0]$ in SA^R
- search the longest prefix that matches the current *content*(i) in SA
- map the content's position in SA to the position of its context (left-part) in SA^R
- compute the distance
- emit the triplet as described before.

In order to get the best matching context/content we look at the predecessor and successor, retrieved by the `bit_ranker` data structure, because they are the elements most similar to the current one.

In Table 3.1 (* means available) there are the context and content dictionaries for the text `swiss␣miss`. Suppose now that we are at character 3, the second "s". At position 6 there is the obvious content matched by a binary search into the suffix array and at position 7 there is the obvious context. Now we proceed in the following manner to get the best matching content (the other case, for the context, is perfectly symmetrical):

- compute the `lcp` between element at position 6 and its predecessor at position 3 (we consider only the available ones)
- compute the `lcp` with the successor element, which is at position 8
- set as best content the one who shares the longest `lcp` (element 8)

The best context is the element at position 10.

	SA		SA^R
1	`␣miss`	1	`␣ssiws`
2	`iss`	2	`im␣ssiws`
3*	`iss␣miss`	3	`iws`
4	`miss`	4	`m␣ssiws`
5	`s`	5*	`s`
6	`s␣miss`	6	`sim␣ssiws`
7	`ss`	7	`siws`
8*	`ss␣miss`	8	`ssim␣ssiws`
9*	`swiss␣miss`	9	`ssiws`
10*	`wiss␣miss`	10*	`ws`

Table 3.1: Context and Content dictionaries.

Continuing the previous example we have to map now the best content found at position 8 into the context dictionary to compute the distance with respect to the data structure illustrated in Table 3.2. Things we know and that could be helpful are:

- position p_{ctn} in T of the best content (3 in the example), that is position $p_{ctn'} = n - 3 - 1 = 6$ in T^R
- position p_{ctx} in T^R of the best context (8 in the example)

Note that we use T^R because the ACB dictionary is based on contexts thus the prediction is based upon them.

All that we need is a function to retrieve, given a position of the text, the position in the suffix array. Call it Φ and define it as follow:

$$\Phi(i) = k \text{ iff } SA^R[k] = i \quad 0 \leq i < n \tag{3.1}$$

So, since we know the position in T of the context and the content we can use Φ to compute the distance

$$\delta = rank(\Phi(p_{ctn'} + 1)) - rank(\Phi(p_{ctx})) \tag{3.2}$$

where the rank function has to be used since there are many unavailable items to skip. In the example we have $\delta = rank(3) - rank(10) = 0 - 3 = -3$.

Since the suffix array library does not provide Φ's computation it is implemented via a simple data structure that access elements by SA values and whose values are barely the indexes of the SA. It is a reverse suffix array. The space consumption of this solution is linear to the input length and correspond to an unsigned int for each character, so we have 4 times the text.

3.2 Decompression

In decompression we have to deal with the problem of keeping sorted suffixes of the decoded text. Here, excluding the idea to develop an efficient data structure for the scope, we have to choice among some tree implementations. Since offsets are performed at each iteration, we opted for a `b+-tree` because it has all the records stored at the leaf level (only keys are stored at interior nodes) and linked together.

SA^R		$content(\Phi(SA^R))$
`swiss␣`	1	`miss`
`swiss␣mi`	2	`ss`
`swi`	$*3$	`ss miss`
`swiss␣m`	4	`iss`
`s`	$*5$	`wiss miss`
`swiss␣mis`	6	`s`
`swis`	$*7$	`s miss`
`swiss␣miss`	8	`$`
`swiss`	9	`␣miss`
`sw`	$*10$	`iss␣miss`

Table 3.2: Mapped data structure for the text `swiss miss`.

Following the Occam's razor principle, for the first version of the software we used an in memory berkeley-db. Since the number of suffixes could be really big we though that such solution will be also good thinking to work on disk for some tests.

Unfortunately we were not able to decompress neither a file of 10MB. After checking out the code by unit tests we did a brute force insertion of the suffixes to check this data structure performance. We obtained very discouraging results since just after few millions insertions the throughput falls until the unusable level on a powerbook G4 1.5GHz with 1.25Gb of RAM. We tried to tune the db but without positive results.

So we have had to start investigating for an appropriate data structure that could handle the problem. Here we briefly illustrate the library we tested and the result obtained.

Data structures comparison We tried the following data structures:

- berkeley-4.7 b+-tree[4]
- hamsterdb-1.0.6[22]

- libredblack-1.3[16]
- luxio-0.1.0[25]
- tokyocabinet-1.4.7[15]
- b+tree implementation given by the team of http://www.scalingweb.com/
- stx-btree-0.8.3[7]
- reverse lexicographically sorted list (Yokoo [27]), our implementation [17]
- gnu libavl-2.0.2[21]

For each one, except for the Yokoo's sorted list, we set as the `db-key` an integer representing the position (`pos`) in the text T and as a comparison function a function that computes the reverse longest common prefix (or the `lcp` depends if we are considering contexts or contents) between the two suffixes identified by the position stored in the `db-key`.

The implementation of the reverse sorted list we made is released under the Gnu GPL3 license and available for download at [17].

It was not our intent to analyze in detail the performance of these data structures so we report only elapsed time and traversal time for some portions of the file english.50MB downloadable at [5].

Table 3.3 shows results obtained to insert all the suffixes and Table 3.4 shows results obtained to traverse the whole data structure one time.

The elapsed time has been calculated using the `clock()` function of the `C` standard library (`stdlib`) so it is a measure of the processor's time.

Looking at the results the `gnu-libavl` library looks as the winner but, after some tests in the field, we decided to use `stx-btree` because in real cases it performs really faster. Also `luxio` has very interesting results but also this library in real cases is slower than the `stx-btree` one. These performance, since construction time is comparable for `gnu-libavl` and `luxio`,

lib	english.1MB	english.5MB	english.10MB
berkeley-4.7 b+-tree	82s	661s	1642s
libredblack-1.3	**8s**	286s	937s
luxio-0.1.0	12s	**219s**	645s
hamsterdb-1.0.6	24s	436s	1310s
tokyocabinet-1.4.7	13s	656s	2520s
gnu libavl-2.0.2 (pavl)	**8s**	**196s**	**594s**
gnu libavl-2.0.2 (avl)	**8s**	**195s**	**590s**
stx-btree-0.8.3	13s	398s	1229s
prefix_list	85s	723s	1680s

Table 3.3: Construction time.

are attributable to the traversal time. This is because in ACB at each iteration we do an offset to get the content to copy and a minimal time difference that does not came out in Table 3.4 becomes the bottleneck since we do two offsets operations at each iteration.

For example to decode a 5MB file `gnu-libavl` is roughly 7 times slower than `stx-bree`.

The decoding algorithm will follow these steps for each triplet read from the input:

- read the startup substring and initialize SA^R keeping a pointer to last item inserted
- find the best context using `prev()` and `next()` methods
- offset of `distance` position
- copy the relative text
- append look-ahead character

lib	english.1MB	english.5MB	english.10MB
berkeley-4.7 b+-tree	1s	8s	15s
libredblack-1.3	5s	191s	∞
luxio-0.1.0	**<1s**	**1s**	**2s**
hamsterdb-1.0.6	<1s	2s	5s
tokyocabinet-1.4.7	2s	21s	43s
gnu libavl-2.0.2 (pavl)	<1s	2s	3s
gnu libavl-2.0.2 (avl)	<1s	2s	3s
stx-btree-0.8.3	**<1s**	**<1s**	**<2s**
prefix_list	**<1s**	**1s**	**2s**

Table 3.4: Elapsed time for a complete traversal.

The substring's copy is done char-by-char as in LZ because in this way it will be possible to copy also a part of the string we are writing. For example if the output buffer is `ABAAB` we can copy 4 characters from position 2 even if we have only 3 characters until the end of buffer. The resulting string will be `ABAABAABA`.

Chapter 4

Experiments

Our experiments are divided into three parts, one investigating the triplets' compressibility, one faced to the time and memory usage of the algorithm and the last one faced to compare these results with some of the other compressors available such as LZOptimal [13], bzip2 [3] and gzip [6]. We also used the compression boosting library [12] for our tests. Before we begin with the discussion on the experiments we briefly recall the main features of the LZOptimal compressor and of the boosting library.

Bit-Optimal Lempel-Ziv [13] compression computes the LZ-optimal parsing of any input string in efficient time and optimal space. Here optimality means to achieve the minimum number of bits in compressing each individual input string, without any assumption on its generating source and for a general class of variable-length codeword encodings.

Technically speaking, they modeled the search for a bit-optimal parsing of an input string $T[1, n]$, as a single-source shortest path problem (shortly, SSSP) on a weighted DAG $G(T)$ consisting of n nodes, one per character of T, and e edges, one per possible LZ77-parsing step. Every edge is weighted according to the length in bits of the codeword adopted to compress the corresponding LZ77-phrase. They consider a class of codeword encoders which satisfy the so called increasing cost property: the larger is the integer to be encoded, the longer is the codeword, such as we have done for ACB.

They provided also a version of the LZ algorithm with unbounded context window which chooses the longest substring to copy located at the closer distance respect to the current position.

Boosting compression library [12] provides a general boosting technique for Textual Data Compression. Qualitatively, it takes a good compression algorithm and turns it into an algorithm with a better compression performance guarantee. It displays the following remarkable properties:

- it can turn any memoryless compressor into a compression algorithm that uses the "best possible" contexts;
- it is very simple and optimal in terms of time;
- it admits a decompression algorithm again optimal in time.

This boosting technique is based upon the Burrows-Wheeler Transform (BWT [24]) such as the bzip2 library. It uses also the Suffix Tree data structure and a greedy algorithm to process them. In their works it is shown that there exists a proper partition of the BWT of a string T that shows a deep combinatorial relation with the k-th order entropy of T. That partition can be identified via a greedy processing of the suffix tree of T with the aim of minimizing a proper objective function over its nodes. The final compressed string is then obtained by compressing individually each substring of the partition by means of the base compressor to boost. Their boosting technique is inherently combinatorial because it does not need to assume any prior probabilistic model about the source T, and it does not deploy any training, parameter estimation and learning.

This library has been used for our tests since `bzip2 -9` is limited to compress the input text in blocks of 900Kb.

File we used to do our tests are all the ones downloadable at [5]:

- english (english texts)
- sources (source program code)
- pitches (MIDI pitch values)
- proteins (protein sequences)
- dna (gene DNA sequences)
- xml (structured text)

We used also a portion of the English web.

Plottings have been made with **gnuplot** and all data manipulation has been made using our tools which also comprehend a version of the Huffman compressor with custom symbol length. We used it to obtain the order-0 entropy of a binary file where symbols were packet not by a single byte as usual but by 4 bytes (the integer we used to store distances and lengths). All the tests have been made on a Powerbook 12" with 1.25GB of RAM except for the memory usage as explained later.

4.1 Plotting results

The most significant part of the emitted data, in terms of compressibility, is represented by the tuple composed by distance and length since the look-ahead character is not correlated to the algorithm itself in any manner. We plotted these distributions in order to understand how ACB performs and to understand if a specific coder could be developed. We also compared these plots with the ones made from the optimal compressor who shares the same tuple, LZOptimal.

Here we first present our results and then the one produced by LZOptimal. In Figure 4.1 the lengths' plot is shown. This distribution is a very narrow gaussian which we could remap to a exponential decay one, since coders work better with such distribution, taking as zero the peak value. We expect also to get the same, or very similar, plot from LZOptimal when we use its greedy version.

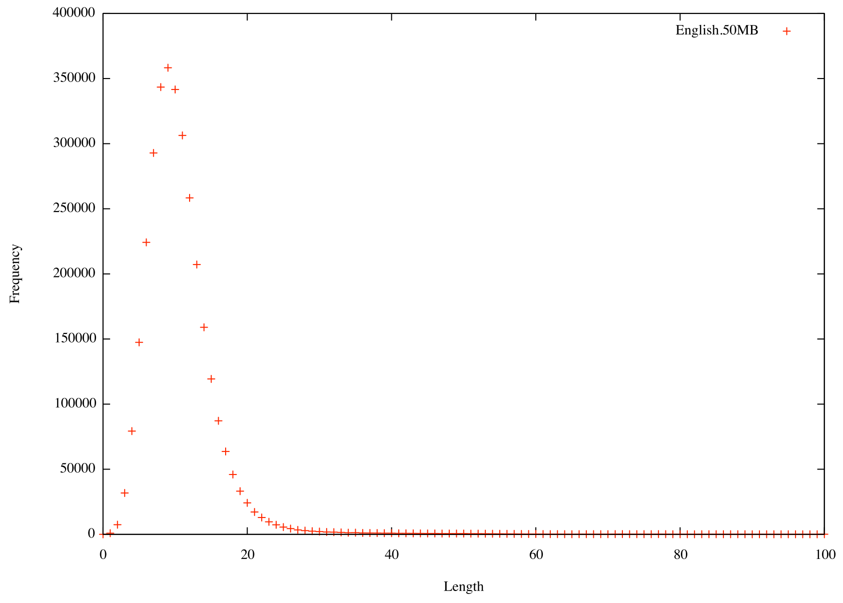

Figure 4.1: ACB's lengths distribution of English text.

In Figure 4.2 is illustrated the distances' plot which is most important to our intents since it is a graphical representation of the main feature of our approach.

The curve at a first glance seems to be good due its skewness, but if we look better we note that the range of distances and thus the range of values to encode is very huge and it spans from the two extremities of the dictionary (which means a distance equals to the length of the read text, equivalent to a copy from the beginning in LZ77). In Figure 4.3 we reported the plot corresponding to the distribution of these big distances.

We expected these values to be limited to a small size window since we expected a good predictor, but in the tests we made we have always a very wide distance range. Moreover, in a relevant percentage of that big values, we copy more than a character as seen in Figure 4.4. In that figure we plotted

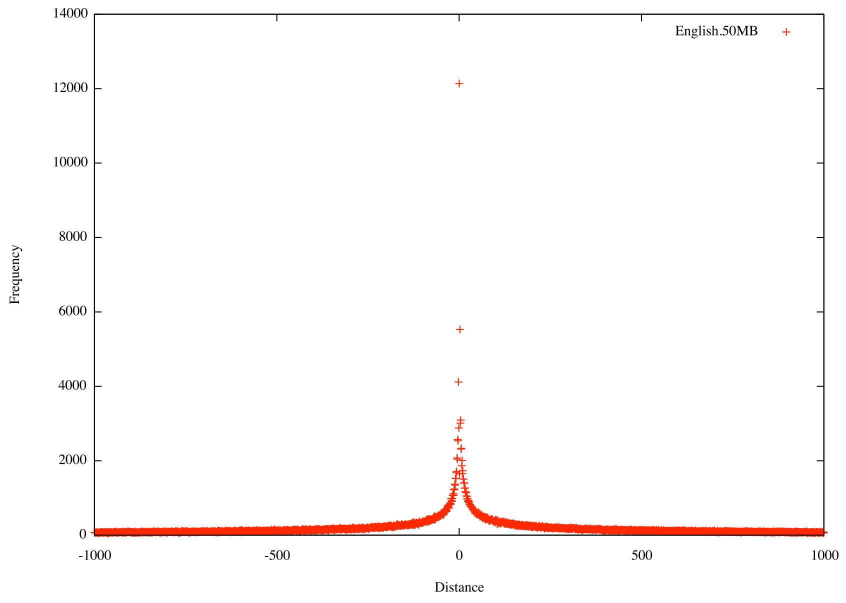

Figure 4.2: ACB's distances distribution of English text.

on x-axe the distance and on y-axe the length of the match to understand how much we copy at that very far place. We noted that also for these limit values, such as 20 millions on a 20MB file, the length of the match is still very good and so we cannot prune these portion of the data in a greedy approach like the one we used. Of course we could parse these couples to emit the optimal tuple, such as in LZOptimal. It is obvious in fact, that if to copy some characters we are going to pay a coding space for the distance comparable to the space of the raw substring, we are only wasting our time. We could also find another way to encode the distances since we have lot of information in the data structure that we are not currently using.

We have to take into account also the following interpretation of the big copy correlated to a big distance. We used a greedy approach so the algorithm instead to take a substring of k characters at a very close distance could choose a substring of $k + 1$ characters at the extremities. We will analyze this correlation later in the section.

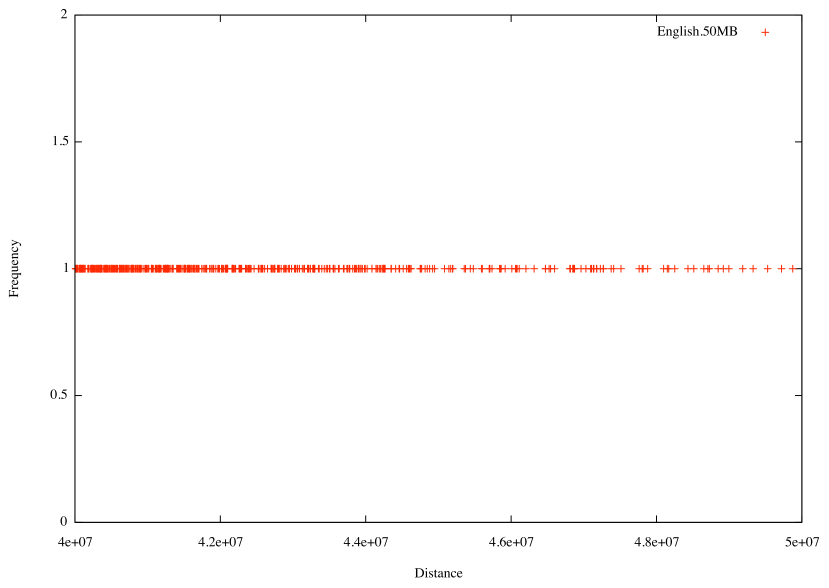

Figure 4.3: ACB's distance distribution of English text for the extremities values.

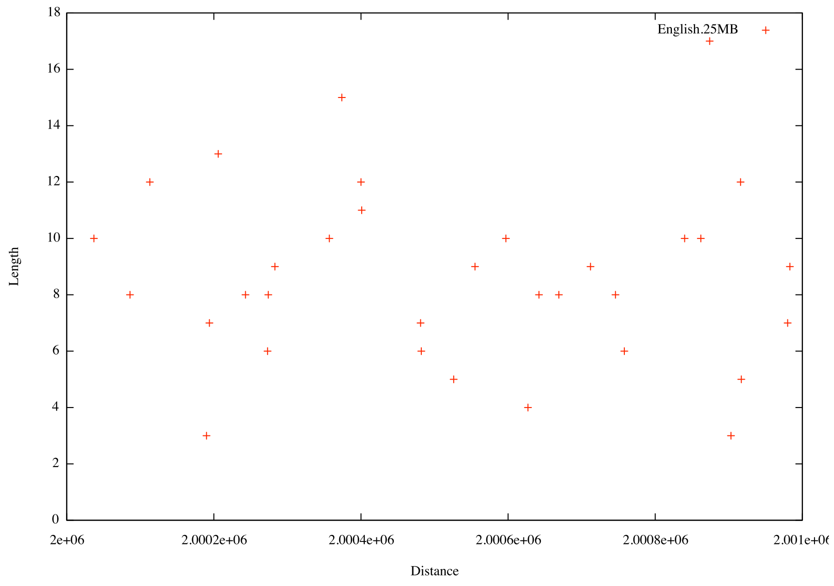

Figure 4.4: ACB's distance with corresponding match length for the extremities values of the English text.

Results obtained by other files are very similar to the given one, except for XML one where the lengths have an oscillating decay probably because of the language tags as shown in Figure 4.5.

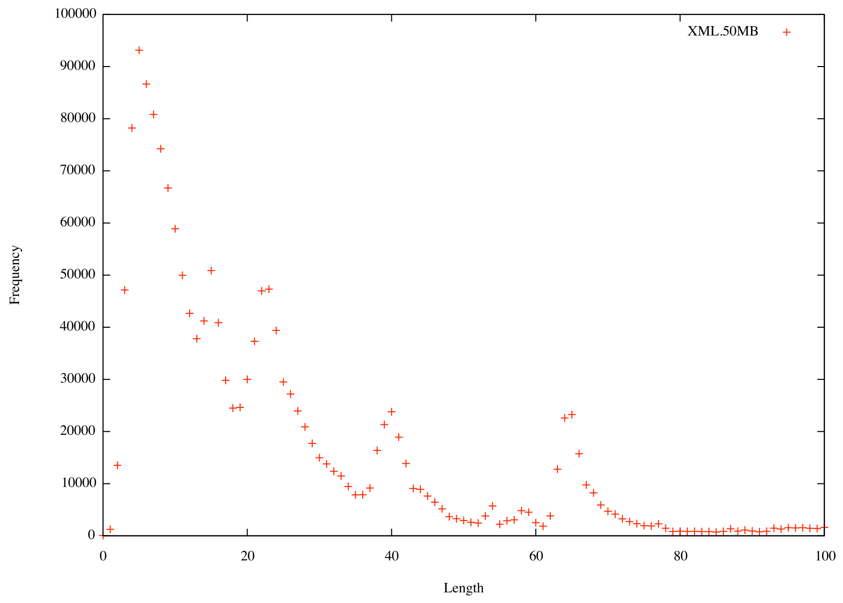

Figure 4.5: ACB's length distribution of XML text.

The same work has been done with the **LZOptimal** version which use the greedy unbounded window settings. We used this version of the software for the plot section because it is the most similar to the ACB's approach. Results are quite similar for the lengths values as shown in Figure 4.6.

However, we expected these result to be identical to the ACB's ones because both compressors use the longest content to copy but, looking at the number of triplets emitted, we noted that ACB has a smaller number of triplets, about 10% on a portion of 10MB of the English file. In practical terms it seems that ACB is able to see longest copies that LZOptimal does not see and we should analyze, but it is not in the intent of our thesis, the LZOptimal code to understand better what is done under the hood. In Figure 4.7 we overlapped the lengths distribution of the two compressors to

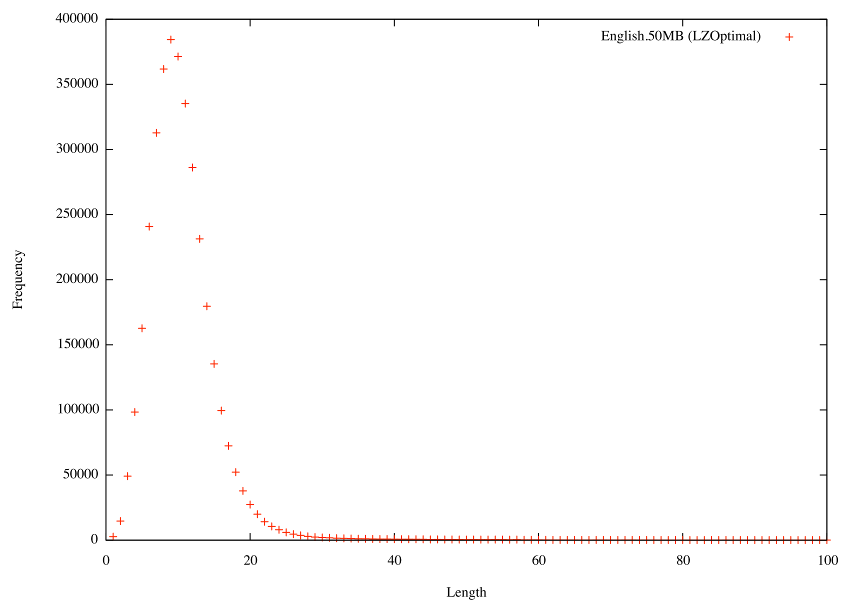

Figure 4.6: LZOptimal's length distribution on English text.

better compare them. ACB has a lower peak value so it has to have a better decreasing curve section, with much more values than LZOptimal for long copies, to justify the smaller number of triplets as showed by the Figure 4.8.

In order to compare the distances' distribution of the LZOptimal with the ACB one we have had to remap the negative distances emitted by out compressor into positives using the following formula:

$$remap(x) = \begin{cases} x = 2 * |x| - 1 & \text{if } x < 0 \\ x = 2 * x & \text{if } x >= 0 \end{cases} \quad (4.1)$$

Then we overlapped the two plots to find similarities as shown in Figure 4.9. From this differential it is very clear that the ACB's distribution is really better because it is highly skewed. Looking also at the extremities values, Figure 4.10, we note that the two plots are almost identical and this means that although these unexpected values ACB is able to generate a better data distribution.

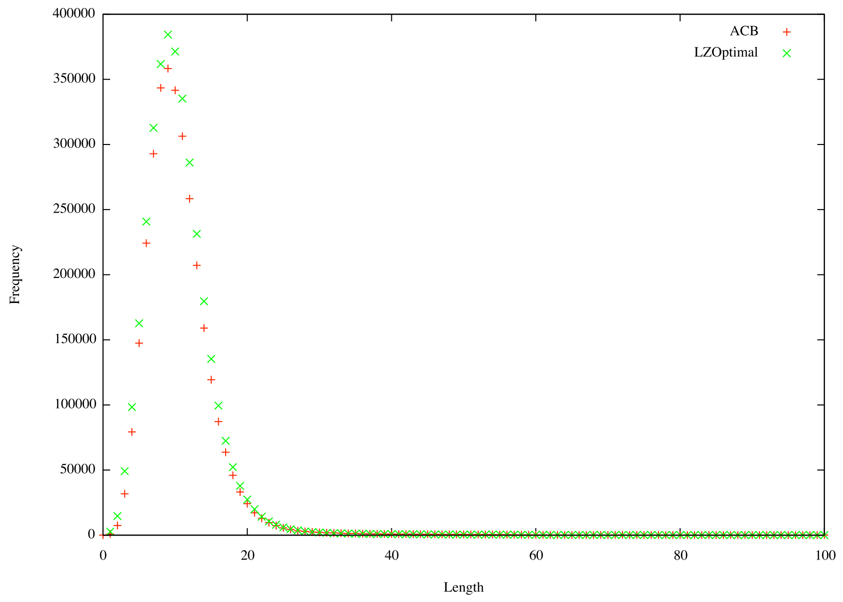

Figure 4.7: Lengths' distribution of LZOptimal and ACB on English text.

The main problem of our implementation is to understand why the distances reach these huge values so that we could think about some tricks to reduce them. What we analyzed is the

- **context error**: the difference, in terms of `bw_lcp`'s length, between the best context used by ACB to compute the distance and the context of the content we are going to copy
- **content error**: the difference, in terms of `lcp`'s length, between the best content copied by ACB and the content associated to the best context.

In practice we would answer the following question: *what if we always copy from the obvious context?* A graphical example of how these deltas are calculated is illustrated in Figure 4.11.

The plots of these absolute errors are shown in Figure 4.12 for the contents and in Figure 4.13 for the contexts.

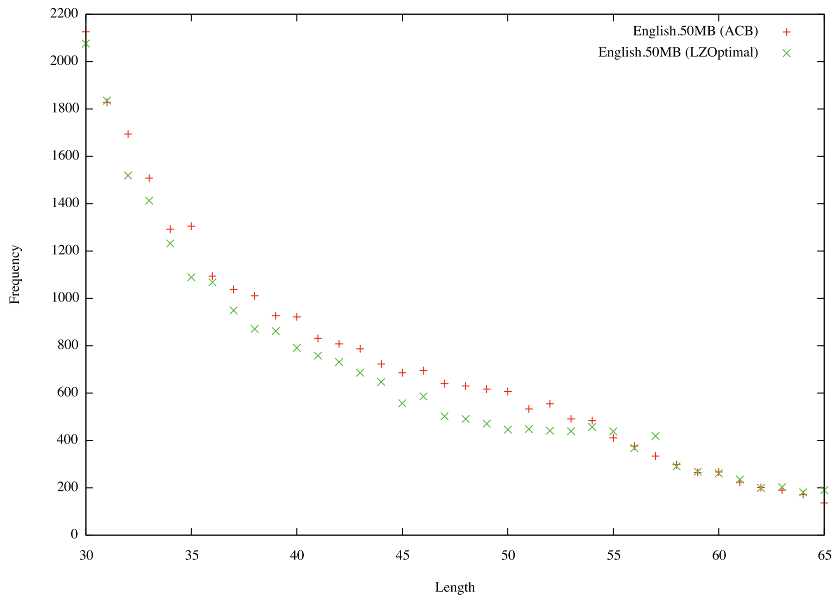

Figure 4.8: Lengths' distribution of LZOptimal and ACB on English text for long copies.

The gaussians produced are not so readable because they represent the absolute error and because there is not present a correlation with the distance emitted. So instead of plotting the distribution of the absolute errors we plotted the distance emitted for such error.

In Figure 4.14 and Figure 4.15 it is shown this relation. We noted that for small deltas we have very big distances so we think that the algorithm can be improved avoiding these dummies copies but the plot has lot of noise and does not take into account if the error is big for the context/content we are considering or not. What evinced from these figures is only the fact that it seems that we emit a very big distance just to copy a character in addition to the one we could copy from the obvious choice. This is obviously a bad situation since the coding space used for the distance is not compensated by the additional character copied.

A brute force approach to avoid these dummy copies could be the one

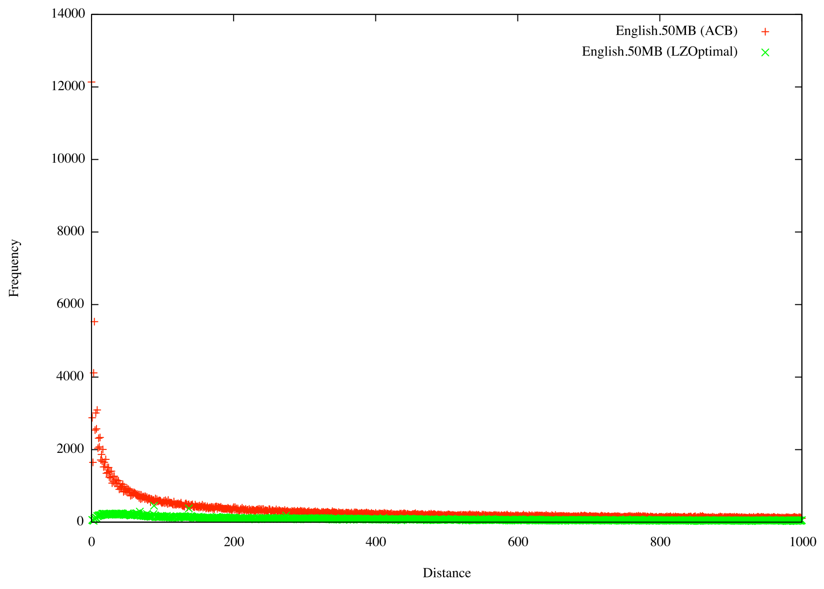

Figure 4.9: Distance distribution of LZOptimal and ACB on English text.

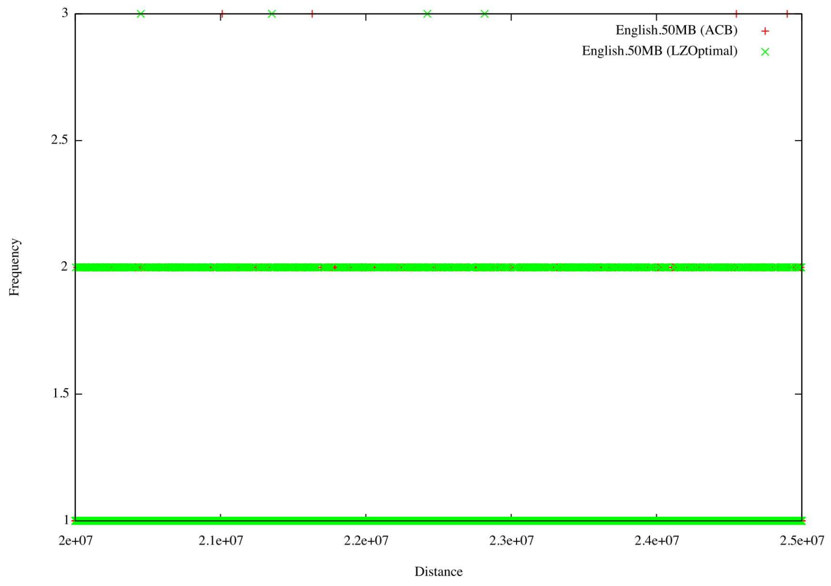

Figure 4.10: Distance distribution of LZOptimal and ACB on English text at the extremities.

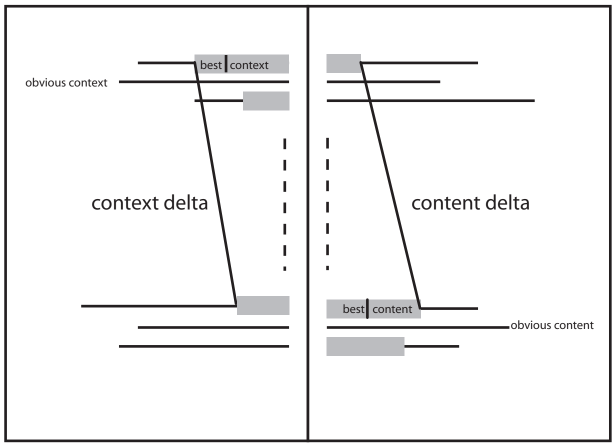

Figure 4.11: Context and content deltas between the obvious and the best one.

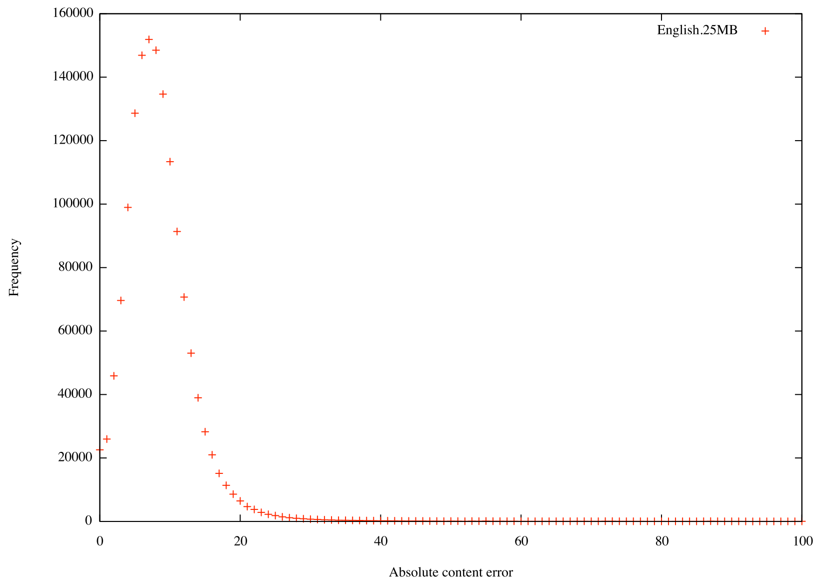

Figure 4.12: ACB's absolute error between best and obvious content length.

which consists in restricting the search set for the best content to only a portion of the whole dictionary. In section 5.1 we reported some tests to compare the theoretically optimal method, used in ACB, with a limited neighborhoods

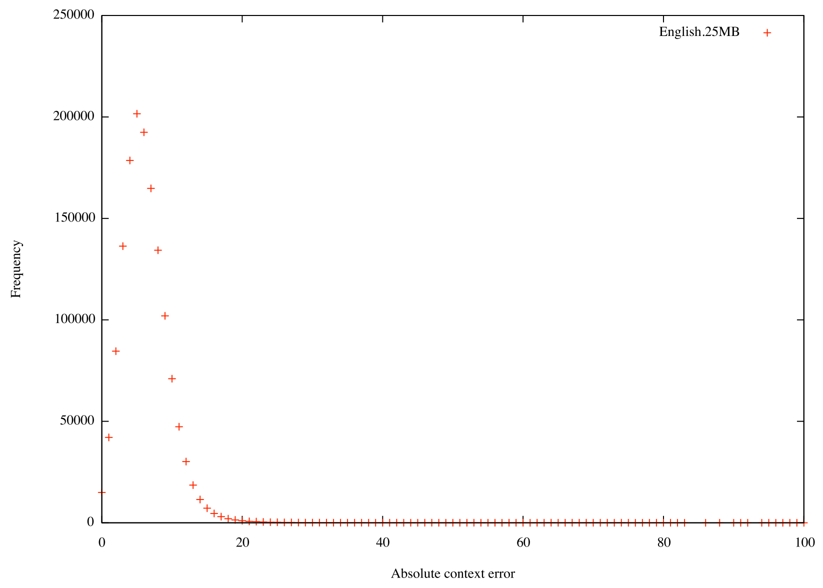

Figure 4.13: ACB's absolute error between best and obvious context length.

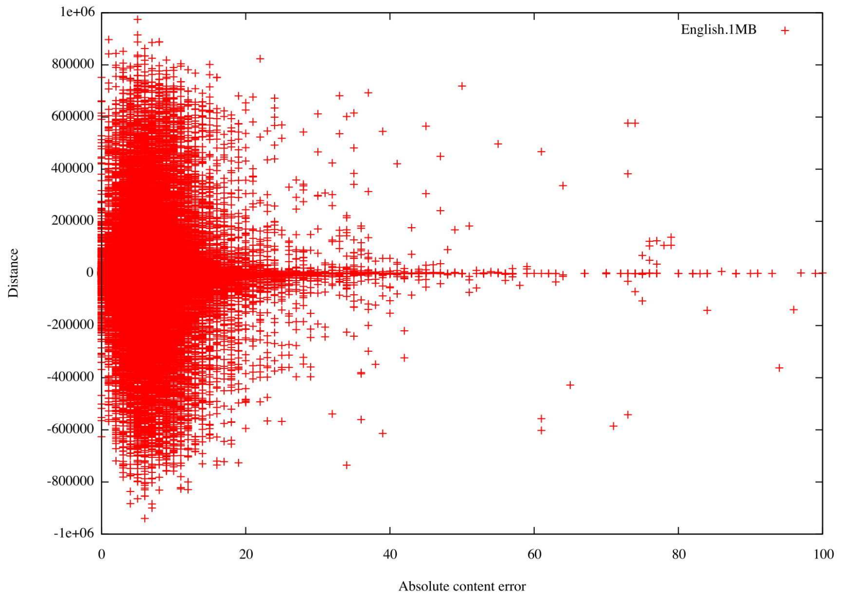

Figure 4.14: ACB's distance corresponding to the content absolute error.

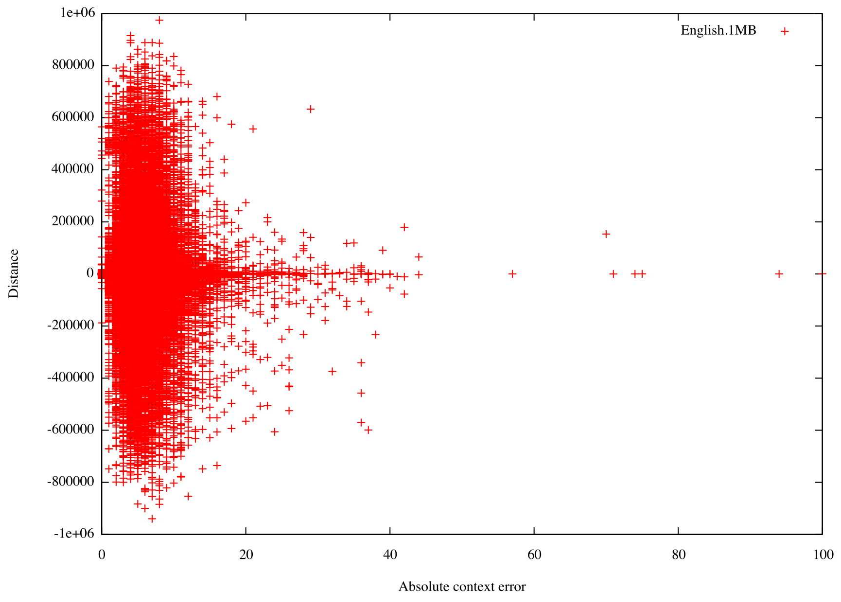

Figure 4.15: ACB's distance corresponding to the context absolute error.

method which could be considered as an heuristic.

The main problem with these error plots, 0s values apart, resides in the incapacity to distinguish if a big distance is generated by a small absolute error on a long context/content or by a small absolute error for a short context/content. To avoid this ambiguity we adopted the relative error which is a better comparing measure for the problem. In fact if we have a content match of 2 characters offsetting will be probably penalizing.

In Figure 4.16 and in Figure 4.17 we present the plot generated using the relative errors respectively for the context and content and here finally we have a really better and understandable plots.

As expected there is no evident relation between the contents' relative error and the corresponding distance. This is because the distance could be interpreted as a function of the prediction, which is made by the contexts and not by the contents. For the contexts in fact we have a very clear relation between the error and the distance. It is interesting to note that we reach

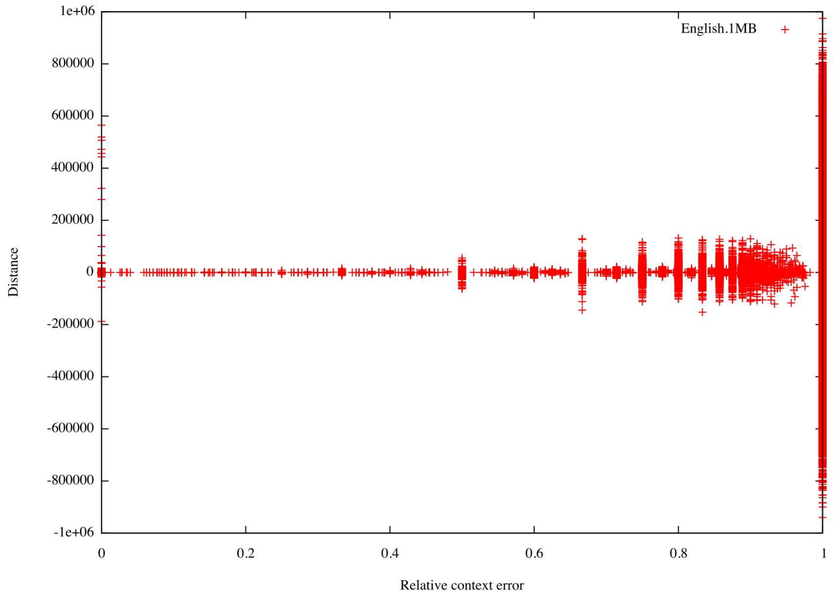

Figure 4.16: ACB's context relative error with corresponding distance.

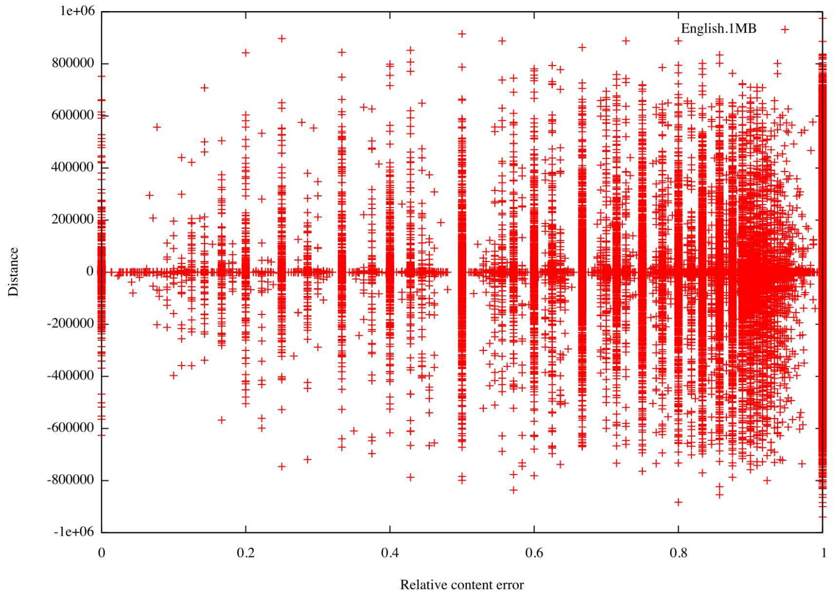

Figure 4.17: ACB's content relative error with corresponding distance.

the very big values, noise apart, only when we have a complete mismatch for the prediction (relative error 1). This means that the prediction accuracy influences very deeply the overall compression since for error values until 0.9 the distance range is very narrow, about 20% of the maximum value emitted.

Since the problem is limited to the cases when we get a complete prediction miss we counted these occurrences and on the 113078 triplets emitted for the english.1MB file they are 51570 which is a big amount of the overall data.

We also tried to correlate the distance with the relative error of contexts and the relative error of contents in a 3D plot as illustrated in Figure 4.18. We see that the content relative error does not seem to be strongly related

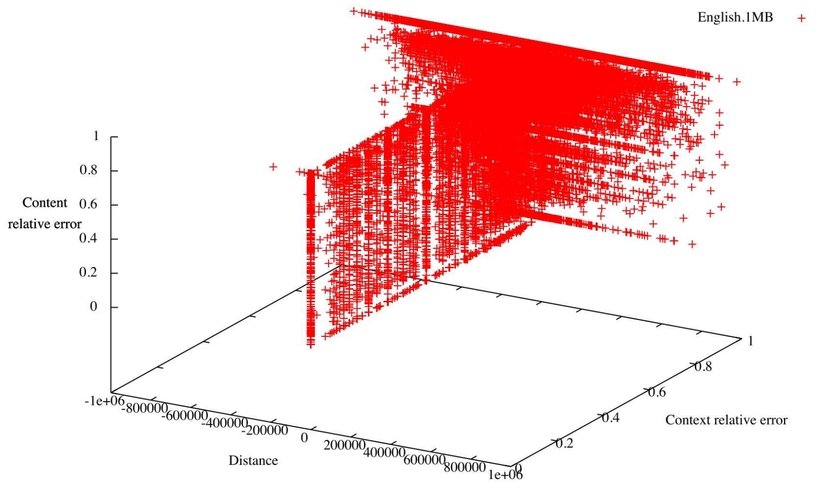

Figure 4.18: ACB's content and context relative errors with corresponding distance.

to the prediction rate since we have the complete range of error values also when we get a correct prediction (context relative error = 0). We notice also that the biggest part of the big distances is located in the high content error zone of the plane defined by context error = 1 which means that if we miss also the content we have a very bad context/content for the prediction, which is obvious.

We found that the bottleneck of our ACB implementation resides in the distances emitted. Now we also know that these distances are emitted only, with a very good approximation, when we have a complete prediction miss. So if we found a method to avoid or limit the context error 1 cases we could increase considerably the compression ratio since we will reduce the amount of different symbols to encode.

Is it true that the amount of the error 1 cases is considerable but is also true that if we try to emit a raw character on a complete miss we could free the path to a good series of predictions. We modified ACB in order to test this hypothesis, get the prediction only when the context matches at least 1 character emitting a raw character otherwise. This naive approach, used just because not to expensive to code, as revealed itself as a bad idea since the number of triplets increases penalizing the overall compression ratio. Of course this is a viable path to future researches since instead of this approach we could use an heuristic to estimate at each step if we will reach better compression using this limited-window technique. In fact we have two cases, one where emitting a raw character instead of a long distance will bring us to a good prediction on the next step with a shorter distance and one where emitting that character will produce a series of 0-copies or a series of bad predictions. This problem could be solved using a weighted graph where each node represent a computation step and where each edge represent the codification's cost. Each node will have 2 edges, one for the ACB greedy version and another for the limited context error ACB version. The same approach could be used in accordion to another kind of encoding such as LZ77, instead of the ACB, which should be better at these cases. What we should take into account is the impact of the escape bit used to discriminate between the two cases. In the test we made for example, the loss to compensate using another distance codification is somewhat like the 4% of the greedy compressed version.

As shown before, differences between LZOptimal and ACB are limited only to the distance distribution at small values. In practical terms we should reach better compression due the skewness of our distribution.

During the experiments we also used the optimal parsing version to compare results. The plots generated are very similar to the one reported before so they are not presented. What we want to report is the interesting capability of ACB to obtain better result, and better compression ratios we hope, than LZOptimal using only the dictionary. This lets opened the door of the weighted graph to create an optimal parsing also for ACB.

4.2 Time performance

As we said before, it was not our intent to give an efficient implementation of this algorithm, so time performance have to be improved. By the way we have some portion of code to refactor that could decrease significantly the computational time, but first we want to understand if it worth.

Table 4.1 illustrates elapsed times for different portion of the English file.

File size in Kb	compression	decompression
498204	4s	17s
2266491	23s	521s
26214400	407s	33153s

Table 4.1: Elapsed time for several portion of English file.

Compression time is practically equal to the LZOptimal's time since to compress 25 megabytes both algorithms employ about 400 seconds, ACB is just a bit slower (5 to 7 seconds). Of course we have to consider that our implementation does not have a triplet encoder. We have not attached one to it because we want first to understand if there exist something ad-hoc, so to the reported time we should add the triplets coding time which should not alter too much the results.

The big problem here resides in the decompression phase where we are not competitive at all, maybe it should be more correct to say that we are not comparable at all. In fact, while LZOptimal decompresses also 25MB in few seconds we employ lot of minutes or hours. This is due to the long distances emitted that in decompression mean lot of offsets within the tree. We have to find a solution to that problem but since the scope of our thesis is to study how ACB performs under the compression ratio we leave it for further developments.

4.3 Memory usage

To compute the memory usage we used the tool `memusage` downloadable at [5]. These tests have been made on a Pentium IV machine with 512MB of RAM since this tool cannot be used on Mac since it uses the `argp.h` library which is a non-standardized `glibc` API extension so we used another machine instead of porting the software.

In Table 4.2 and in Table 4.3 we reported the ACB and LZOptimal results.

Compression						
File size in Kb	heap total		heap peak		stack peak	
	ACB	LZOptimal	ACB	LZOptimal	ACB	LZOptimal
1048576	29817861 bytes	23107154 bytes	27693667 bytes	22740984 bytes	8208 bytes	2864 bytes
10485760	460095354 bytes	307717815 bytes	283816816 bytes	227390868 bytes	13408 bytes	6496 bytes
26214400	1240661227 bytes	804452246 bytes	724457933 bytes	568474008 bytes	13056 bytes	6240 bytes

Table 4.2: ACB and LZOptimal memory usage to compress several portion of the English file.

As happened for the time performance also for the memory consumption we have comparable results, although worst, for the compression phase while in decompression, due to the data structure we have to keep in memory, we have a bigger consumption. We have also to mention that in LZOptimal during the decompression phase there is no overhead for the copy thus the only thing to do is a subtraction (to get the position within the decoded

Decompression						
File size in Kb	heap total		heap peak		stack peak	
	ACB	LZOptimal	ACB	LZOptimal	ACB	LZOptimal
1048576	28273794 bytes	1403080 bytes	28273442 bytes	1402728 bytes	1568 bytes	284 bytes
10485760	201698970 bytes	12808877 bytes	194610634 bytes	12808525 bytes	1712 bytes	284 bytes
26214400	492427826 bytes	31315754 bytes	462420210 bytes	31315402 bytes	1712 bytes	284 bytes

Table 4.3: ACB and LZOptimal memory usage to decompress several portion of the English file.

text) and an access to that portion of the text (which is also computed in constant time and space). In ACB on the contrary we have to rebuilt the same dictionary used in compression. We think that it is not fair to compare the decompression phase of the two compressors since they are too different. We reported the values just to a mere confront.

4.4 Compression ratio

To compute the final compression ratio of ACB we used the order-0 entropy (H_0) to encode the emitted triplets, so it is like we choose a range encoder. We have not attached a coder to our ACB implementation since we would first analyze results in order to find if there is a specific coder to use instead of waste time adapting an existing one which probably will be discharged. By the way the H0 is a reliable measure of the compressed size because using a range coder the loss from the optimal result is really minimal. We compared our results with some of the best known compressors such as LZOptimal, gzip (-9 option), bzip2 (-9 option) and its boosted version.

Table 4.4 shows compression ratios.

In the tests we used both the LZOptimal versions, greedy and optimal, because the first one is useful to understand how much we gain changing only the distance interpretation while the latter one is useful to understand if we are still competitive when an optimal policy is applied. Also gzip is comparable in terms of emitted data but since it has a limited window size

File	gzip -9	bzip2 -9	LZOpt (greedy)	LZOpt (optimal)	bst -fm -a6	bst -fm -a2 -y16384 -z64	bst -f -a7 -y65536 -z256	ACB (ideal)
english.1MB (1048576 bytes)	380327 (0.36)	286148 (0.27)	377740 (0.36)	353224 (0.33)	**282239 (0.26)**	**282905 (0.26)**	**281347 (0.26)**	332422 (0.32)
english.10MB (10485760 bytes)	3950985 (0.37)	3012397 (0.28)	2559363 (0.24)	2321837 (0.22)	2319811 (0.22)	2300303 (0.22)	**2214666 (0.21)**	2272364 (0.21)
english.25MB (26214400 bytes)	9828313 (0.37)	7522109 (0.28)	5697944 (0.22)	5100074 (0.19)	5321362 (0.20)	5277340 (0.20)	**5046839 (0.19)**	**5063542 (0.19)**
dblp.xml.1MB (1048576 bytes)	183931 (0.17)	115871 (0.11)	174088 (0.16)	164728 (0.15)	**112870 (0.10)**	113601 (0.10)	116469 (0.11)	143848 (0.13)
dblp.xml.10MB (10485760 bytes)	1825535 (0.17)	1174240 (0.11)	1581552 (0.15)	1455734 (0.14)	1047196 (0.09)	**1036605 (0.09)**	1045184 (0.09)	1375252 (0.13)
dblp.xml.25MB (26214400 bytes)	4517302 (0.17)	2937212 (0.11)	3794954 (0.14)	3452262 (0.13)	**2522973 (0.09)**	**2486327 (0.09)**	**2487936 (0.09)**	3326610 (0.12)
dna.1MB (1048576 bytes)	285687 (0.27)	274488 (0.26)	329713 (0.31)	301554 (0.29)	269030 (0.25)	**255507 (0.24)**	267874 (0.25)	**255549 (0.24)**
dna.10MB (10485760 bytes)	2836601 (0.27)	2723685 (0.26)	3162993 (0.30)	2832127 (0.27)	2618141 (0.25)	2484962 (0.23)	2594753 (0.24)	**2446727 (0.23)**
dna.25MB (26214400 bytes)	7096159 (0.27)	6815089 (0.26)	7800001 (0.29)	6965414 (0.26)	6494091 (0.25)	6164332 (0.23)	6425845 (0.24)	**6035427 (0.23)**
sources.1MB (1048576 bytes)	255409 (0.24)	198528 (0.19)	246294 (0.23)	234480 (0.22)	**193683 (0.18)**	**195942 (0.18)**	201759 (0.19)	221592 (0.21)
sources.10MB (10485760 bytes)	2403782 (0.23)	1971044 (0.18)	2228068 (0.21)	2048697 (0.19)	**1852874 (0.17)**	**1859264 (0.17)**	1863561 (0.17)	2107311 (0.20)
sources.25MB (26214400 bytes)	6177216 (0.30)	5233647 (0.19)	5672347 (0.21)	5131207 (0.19)	4801310 (0.18)	4813679 (0.18)	**4750516 (0.18)**	5388999 (0.20)
web (26214400 bytes)	5249476 (0.20)	4168425 (0.16)	2922143 (0.11)	2657049 (0.10)	**2463698 (0.09)**	**2469670 (0.09)**	**2476088 (0.09)**	2627404 (0.10)
proteins.1MB (1048576 bytes)	402689 (0.38)	420336 (0.40)	435015 (0.41)	414073 (0.39)	419691 (0.40)	418008 (0.39)	413619 (0.39)	**340820 (0.32)**
proteins.10MB (10485760 bytes)	3547759 (0.33)	3583043 (0.34)	3646708 (0.34)	3371309 (0.32)	3282741 (0.31)	3273363 (0.31)	3216026 (0.30)	**2940161 (0.28)**
proteins.25MB (26214400 bytes)	10308574 (0.39)	10232402 (0.39)	9123674 (0.34)	8337249 (0.31)	8633151 (0.32)	8609256 (0.32)	8490721 (0.32)	**7339947 (0.27)**

Table 4.4: Compression results comparison.

in which to choose the best content to copy it will be not fair to use it as a baseline.

Our results are always better than the LZOptimal with greedy parsing ones and this is a confirm of the effectiveness of the ACB method, the distance interpretation involves a great improvement. A more encouraging result comes out when we look at the LZOptimal with optimal parsing. Here we do better in most test cases which is a very interesting thing because we use only the dictionary without any parsing optimization which should also be applied to ACB.

Among all compressors our results are always really competitive and in most cases really better such as in the surprising ratios reached for the DNA and the PROTEINS sequences where compression is still an open research problem and where context-based compressors are not usually the best choice. A further analysis will surely be to investigate what we can do to improve compression ratio on these fields.

4.5 Results

In this chapter we analyzed ACB under many aspects and now we have a clearer idea of what to do for further developments. We departed from a scratch, from an informal interpretation, and we came out after lot of modifications and analysis with something that has been deeply studied in order to extrapolate as many information as possible.

Results we obtained are very good and justify future work in order to improve space/time efficiency to create a production version of the software. Also in compression ratio we think that some improvements may be possible, specializing ACB for the particular case of the PROTEINS/DNA or finding an ad-hoc coder for example.

What we have showed is that the method has great potentialities and, as always when you get something good, there are also many issues to deal with. First of all the distance encoding on the compression side and the complex data structure used for the dictionary on the time/space side. The developed version in fact should be considered only under the academic aspect since the elapsed time is too big for an end user tool, especially if we consider the decompression phase. Of course after the distances problem will be solved, if there exist a solution, this issue will be partially solved too.

Some of the things we could try in order to achieve these objectives, without having to alter too much the algorithm written here, are the followings:

- **reduce the search set of contexts**. This is a brute force approach to limit the maximum distance emitted. Since we think that many long distances are generated just to copy a substring a little bit longer than the one which could be copied departing from a closer distance, this approach should be viewed as an heuristic to limit this case which reduce also complexity.

- **reduce the content order**. This is another way to limit the big offset paying a shorter match length gain. If we set it to 1 character ACB will be equivalent to the $(\alpha, \beta) - HYZ$ implementation of Yokoo with another distance interpretation.

- **reduce the context order** (it will introduce duplicates). As for the substring to copy we also think that in many cases when we search the best context, since we always take the longest, we choose a context which is not the optimal one just because of the longer match due to the greedy approach. The duplicated can be treated using a queue, a priority queue or any other policy which improves the probability to get the first item as the right choice. This method introduces another element to the output which refers to the duplicate to choose, we call it the ranking.

In Chapter 5 we report such experiments and we propose some new ideas about how we think things could be done. Note also that we have improvement margins on lengths and look-ahead characters since there is so much unused information within the dictionary.

An example of this unused information is the delta between the longest common prefix between the look-ahead buffer and the previous/next content. This value could be subtracted from the length of the match since it is available also in compression in order to obtain a better lengths' distribution.

Chapter 5

New ideas

ACB performs very well in the experiments we made and in this chapter we are going to describe some methods we could try to improve it. The points to work on are faced one to increase compression ratio and one to decrease computation time.

On the first area we have

- find a method to reduce the emitted lengths
- find a method to reduce the emitted distances

and on the second area we have

- find how much does it cost in compression ratio to reduce the search set within we will search the content to copy
- the impact of limiting the longest common prefix during compression

In this section we present the first area enhancements while the others will be presented later.

To reduce the range of values' **lengths** we begin from the following idea:

if we are at position pos in T and we have to jump x items from a position p in the SA to copy a substring, it means that $\forall y$ such that $y \in]p-x, p+x[$

$$\textit{we have}$$
$$lcp(content(pos), content(SA[p+x])) > lcp(content(pos), content(SA[y])).$$

The proof is really simple. We always copy the content which is at the shortest available distance. Suppose now that there exist an element, say k, which is closer than x and that has the same prefix we want to copy from $p + x$. By the assumption done before we have that $x = k$ and so that such $k < x$ can not exists.
So, instead of emitting the whole length, we could emit the difference between the full length of the match and the longest common prefix computed among all the suffixes in the range. For example in Table 2.5 instead of emit 6 we could emit 4 due the longest common prefix is 2 (with item at position 8).

For the **distances** since we have to uniquely identify the item, the situation is trickier. Some of the possibility we thought about are:

- as in the Yokoo's works counting the number of distinct symbols until the one who matches the first character of the content to copy and then, counting the number of other contents to skip until the matched one. This will introduce another element to the output which is what we have called ranking. In Table 2.5 we will have a 0 meaning that the content to copy starts with an `s` and a 3 to indicate that is the third content who starts with an `s` that we have to take.

- acting like in the Shannon's work, emitting a stream of bits representing correct or incorrect guesses for a character and moving to the next character at each correct guess. In practical terms beginning from the best matching context (`ctx`) we act in the following manner assuming we are at position `x = 0` and at position `ctn` for the best matching content:

```
while (x < match_length)
  {
    if (ctx == ctn) then // we reached the correct item
```

```
            emit_escape_sequence();

          if (content(ctx)[x] == content(ctn)[x]) then
            {
              emit_1();
              ++x;
            }
          else
            {
              emit_0();
              skip_to_next_not_seen_symbol();
            }
        }
```

In Table 2.5 we will have the sequence `10010` and the escape sequence. The first one means that the `s` is correct while the two zeros means that the second `s` and the next not seen symbol `i` are not correct. The fourth 1 indicates the correct guess for the space while the last 0 indicates the incorrect `i`. The risk we could encounter on using this approach is that we could emit a sequence really big if the predictor is not a good predictor since we have deep correlations with alphabet size and guessing rate. The worst case is when we iterate `match_length` times and for each iteration we skip all alphabet symbols (call it $|\Sigma|$), in formula

$$O(|\Sigma| * match\_length) \tag{5.1}$$

Now we present a method who needs bigger modification of the algorithm but that should be able to solve many issues. Before we have spoken about the rank as a measure to indicate how many items which starts with a given symbol we have to skip. Another and better way to interpret this concept could be the following. In all the cases we considered the best content has been always identified by the number of items to skip without keeping in consideration the probability that the given content has in respect to the others. To clear up we give an example. Suppose we have the following dictionary generated from the text `ababab|ab`, where as always the look-ahead buffer is the substring after the pipe:

```
     a | babab
   aba | bab
 ababa | b
    ab | abab
  abab | ab
ababab | $
```

In this case the conditional probability to have an `a` content given a context who starts with `b` is equal to 1. The same occurs if we consider the content `ab`.

We could order the contents in the following manner:

- take the subset of contexts whose `bw_lcp` with the obvious one is equal to 1
- find the longest common prefix, the `match_len`, between the look-ahead buffer and the contents within this subset
- associate to each content the frequency of its substring of length equals to `match_len`

In our example we will have:

- context_subset = {`ab`, `abab`}
- `match_len` = 2
- freqs_set = {{`ab`, 2}, {`abab`, 2}}

The output will be the triplet `<0,2,eof>` where the 0 indicates the first item in the ordered freqs_set.

We could also extend the context-order used to create the subset to a greater value or to the longest common prefix between the order-1 subset (`ab` in the example) to increase the model accuracy.

In Chapter 4 we spoked about a technique to avoid the context relative error 1. We used one additional bit per triplet to discriminate to the method we were using to encode the current distance, ACB vs LZ for example. The bit could be removed if we assume, but first we have to do some tests to understand the new curve fitting, that on context relative error 1 we automatically choose LZ77 distance.

5.1 jcb

This compressor is a variant of ACB, developed in order to understand the impact onto the compression ratio of the enhancements described before for the computation time area. It uses only one data structure (the `gnulibavl`) and works within a limited context dictionary, what we have called the neighbourhood. It is also customizable for the context/content order. When we limit the number of characters to compare to lexicographically sort the data structure we are going to introduce duplicates entries. The default policy applied is to keep them into a LIFO queue, where not differently indicated, and to add an element to the output stream to indicate the rank of the chosen content within this list.

We developed it in the following versions:

- base. It performs such as ACB.
- without look-ahead character. We emit the look-ahead character only if the match length is 0.
- using a priority queue to treats contents associated with a duplicated context

- distance encoded counting the number of distinct symbols and then offsetting within them. This is the method we exposed before when we were speaking about the distance reduction.

In Table 5.1 are shown result obtained using jcb base with unlimited context and content length, at varying of how many dictionary's items we look upright and downright to the obvious context. These tests are faced to understand how much the context dictionary size influence the overall compression.

File	ACB	jcb -n2	jcb -n32	jcb -n128	jcb -n512	jcb -n1024
english.1MB (1048576 bytes)	332422 (0.32)	486569 (0.46)	388928 (0.37)	367052 (0.35)	353495 (0.33)	348765 (0.33)
dna.1MB (1048576 bytes)	255549 (0.24)	418182 (0.39)	330477 (0.31)	312213 (0.29)	300813 (0.28)	296359 (0.28)

Table 5.1: Compression results of jcb with different neighbourhood size.

We notice that for a reasonable small context dictionary neighbourhood, such as 1024, we reach a compression ratio which is only 1% worst than the best one generated considering the whole data structure. This means that the majority of distances for a good compression ratio are very close to the obvious context as also shown in Figure 4.16. We think that the difference is due to long substrings which are copied from the extremities as shown in Figure 4.4. This is a remark that indicates how the greedy approach used for ACB is not the best one. We should rethink the method used to choose the best content or at least we should try the graph optimization of LZOptimal.

Shorter distances does not only mean a better compression ratio but also a really faster decompression phase. The offsets are computed by iterating the data structure and not by direct access to the element as we could have done in a suffix array, so each distance is equivalent to a linear scan of the tree and this is unacceptable for the data we produce. This is why the decompression phase is so slow. We have to consider that we did not optimize the `b+-tree` in order to get a faster offset, things that can be done taking the count in the inner node of the number of leafs contained in the subtree for example.

Some tests have been made also using the other versions of jcb but the results were always worst or nearly equals to the ACB ones. This is a confirm that the method we used to compute the distance is reliable and that naive variations does not show any improvement path.

Chapter 6

Conclusion

In this thesis we illustrated the ACB compression method and we showed its real performance. The literature is poor of results about this compressor so we are nearly the first producing such deep analysis and explaining in detail the implementation choices adopted. We proved that the intuition behind ACB was correct and that this method has a great potential.

This does not mean that we have reached a solution, there are some issues of this approach which should be better understood. For example, the succinct encoding of the distances: we believe that one should deploy, at its full extent the structure information present in the dictionary. We consider this work as starting point for future analysis and implementations faced more on efficiency than on efficacy.

Another interesting aspect that has to be investigated is relative to the DNA and PROTEINS sequences where we obtained really surprising results. We should depart from ACB and understand how to specialize it for this particular case.

Future works should take into consideration also the development of a new data structure to be used as dictionary which implements a fast offset method in order to obtain a time acceptable decoding procedure.

Thinking about the boosting library work [12] another view of the problem could be answering the following question:

Does it exist a partition of the input text which has the property of optimality for a disjoint-block ACB compression?

In other words, we would like to know if is it possible to partition the text and compress each partition separately from the others achieving possibly better results of compressing the whole text at once. In positive case we will be able to compress files of any size breaking the space/memory limit of this implementation. We could also implement a parallel version of the algorithm since the assumption foresees disjoint blocks.

Bibliography

[1] Charles bloom on acb. http://www.cbloom.com/news/leoacb.html.

[2] George buyanovsky on acb. http://www.cbloom.com/news/bygeorge.html.

[3] The libbzip2 library. http://www.bzip.org/.

[4] The oracle berkeley db library. http://www.oracle.com/technology/products/berkeley-db/index.html.

[5] Pizzachili corpus compressed indexes and their testbeds. http://pizzachili.di.unipi.it/.

[6] The zlib library. http://www.zlib.net/.

[7] Timo Bingmann. Stx b+ tree c++ template classes. http://idlebox.net/2007/stx-btree/.

[8] George Buyanovsky. Associative coding (in russian). *Monitor*, 8:10–19, 1994.

[9] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[10] Peter Fenwick. Symbol ranking text compression with shannon recodings. *Journal of Universal Computer Science*, 3(2):70–85, Feb. 1997.

[11] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[12] Paolo Ferragina, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:2005, 2005.

[13] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of lempel-ziv compression. In Claire Mathieu, editor, *SODA*, pages 768–777. SIAM, 2009.

[14] Yehuda Hershkovits and Jacob Ziv. On sliding-window universal data compression with limited memory. *IEEE Transactions on Information Theory*, 44(1):66–78, 1998.

[15] Mikio Hirabayashi. Tokyo cabinet: a modern implementation of dbm. http://tokyocabinet.sourceforge.net/.

[16] Damian Ivereigh. Redblack balanced tree searching and sorting library. http://libredblack.sourceforge.net/.

[17] Jonathan Masci. Implementation of the yokoo's prefix list described in [27]. http://www.google.com/p/prefix-list.

[18] Yossi Matias, S. Muthukrishnan, Süleyman Cenk Sahinalp, and Jacob Ziv. Augmenting suffix trees, with applications. In *ESA*, pages 67–78, 1998.

[19] Alistair Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Information Theory*, 38(11):1917–1921, 1990.

[20] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[21] Ben Pfaff. Gnu libavl. http://www.stanford.edu/ blp/avl/.

[22] Christoph Rupp. Hamster db. http://hamsterdb.com/.

[23] C.E. Shannon. Prediction and entropy of printed english. *Bell Systems Technical Journal*, 30:50–64, 1 1951.

[24] A Block sorting Lossless, Michael Burrows, M. Burrows, David Wheeler, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[25] Hiroyuki Yamada. Lux io - yet another fast database manager. http://luxio.sourceforge.net/.

[26] H. Yokoo. An adaptive data compression method based on context sorting. *Data Compression Conference*, 0:160, 1996.

[27] Hidetoshi Yokoo. A dynamic data structure for reverse lexicographically sorted prefixes. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99), LNCS 1645*, pages 150–162, 1999.

[28] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.