Implementation of a data compressor based on the AEP Theorem

A consequence of Shannon's AEP theorem is the possibility of constructing a very straightforward compression algorithm, able to reach entropy for i.i.d. sources. This thesis describes a possible implementation of such algorithm, that follows closely the ideas expressed by the AEP theorem, and analyses the tradeoffs made in choosing its data structures and techiniques. It also provide some comparison benchmarks, on common corpuses, against the well known gzip and bzip2 compression programs.

Emanuele Acri

Mat. 202623

Supervisor:

Filippo Mignosi

A thesis presented for the degree of:
Bachelor Degree in Computer Science (F3I)

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica.
Università degli Studi dell'Aquila.

1 Luglio 2016

Implementation of a data compressor based on the AEP Theorem

A consequence of Shannon's AEP theorem is the possibility of constructing a very straightforward compression algorithm, able to reach entropy for i.i.d. sources. This thesis describes a possible implementation of such algorithm, that follows closely the ideas expressed by the AEP theorem, and analyses the tradeoffs made in choosing its data structures and techiniques. It also provide some comparison benchmarks, on common corpuses, against the well known gzip and bzip2 compression programs.

Emanuele Acri

Abstract

In his seminal 1948 paper [1], which gave birth to Information Theory as a new field of study, Claude Shannon proven an important theorem known as AEP (Asymptotic Equipartition Property Theorem). A consequence of this theorem is the possibility of a data compression algorithm for sequences of symbols generated using an i.i.d. source, able to map such sequences into binary strings having a length which converges to the entropy of the original sequence under certain conditions.

The compression algorithm proposed by Shannon is straightforward, but perhaps more suitable to be used in a theoric proof that to be implemented on a real machine. The difficulties arise from the division between typical and non-typical sequences that the algorithm operates. Since finding all possible sequences is a combinatorial problem, the quantity of data the algorithm must manipulate is considerable, especially when using a simple approach such as enumerating and sorting all the sequences.

However, in a real scenario, it is more common to compress a file, which is a finite-length string of symbols. This allows an implementation of a compression algorithm to avoid most combinatorial problems, relaxing some theoretical assumptions.

In this document we follow this second approach, focusing on an acceptable algorithm able to operate in the real world. A brief description of the combinatorics of the problem is included in Appendix B.

The first section of the document is dedicated to the fundamental definitions and theorems of information theory, and may be skipped by a person familiar with the subject.

Contents

Entropy and Basic Definitions

To lay a foundation for subsequent theory, this chapter summarizes some basic definitions and concepts of information theory. As a reference guide we used the classical textbook by Cover and Thomas [2].

The concept of *entropy* is fundamental for the entire discipline of information theory, since it acts as a measure of information in accord to both our intuitive notion of information and to some mathematical properties we would like information to have.

However, the concept of information is very broad and varies depending on the prior knowledge we have of the process to analyze and model.

We restrict ourselves to the most general definition of entropy:

Definition (Entropy)**.** Let X be a discrete random variable with alphabet χ and probability mass function $p(x) = Pr\{X = x\}$ with $x \in \chi$.

The *entropy* $H(X)$ of this discrete random variable is defined by:

$$H(X) = -\sum_{x \in \chi} p(x) \log p(x)$$

Note that in this document we will always use a base 2 for the logarithms, since we are interested in a measure of information expressed in bits.

AEP and the Shannon Compressor

Again we need to borrow some definitions from probability.

Definition (Convergence of random variables, in probability)**.** Given a sequence of random variables $X_1, X_2, \ldots$ we say that the sequence converge to a random variable X *in probability* if for every $\epsilon > 0$ we have $Pr\{|X_n - X| \geq \epsilon\} \to 0$ for $n \to \infty$.

Theorem (AEP)**.** If $X_1, X_2, \ldots$ are i.i.d. random variables having probability mass function $p(x)$, then

$$-\frac{1}{n} \log p(X_1, X_2, \ldots, X_n) \to H(X)$$

in probability.

Proof. Function of indipendent random variables are also indipendent random variable. Thus, since the X_i are i.i.d., so are $\log p(X_i)$. Hence, by the weak law of large numbers,

$$\begin{aligned} -\frac{1}{n} \log p(X_1, X_2, \ldots, X_n) &= -\frac{1}{n} \sum_i \log p(X_i) \\ &\to -E \log p(X) \quad \textit{in probability} \\ &= H(X) \end{aligned}$$

which proves the theorem.

□

We can now imagine the set of all possible sequences of length n, of form $X_1, X_2, \ldots, X_n$, where $X_1, X_2, \ldots, X_n$ are i.i.d., and the probability of a given sequence is $p(X_1, X_2, \ldots, X_n)$.

The AEP theorem allow us to divide the set of all possible sequences into two diffent sets: the *typical* set, containing sequences with probability close to the entropy; and a *nontypical* set, containing all the other sequences.

Any property that is proven for the typical set will the be true with high probability for the whole set of sequences, and will determine the behaviour of a large sample.

These concepts are captured by the following definition and theorem.

Definition (Typical set)**.** The *typical set* $A_\epsilon^{(n)}$ with respect to $p(x)$ is the set of sequences $(x_1, x_2, \ldots, x_n) \in \chi^n$ with the property

$$2^{-n(H(X)+\epsilon)} \leq p(x_1, x_2, \ldots, x_n) \leq 2^{-n(H(X)-\epsilon)}$$

As a consequence of the AEP we can show that the set $A_\epsilon^{(n)}$ has the following properties:

Theorem (Properties of the typical set)**.** :

1. If $(x_1, x_2, \ldots, x_n) \in A_\epsilon^{(n)}$, then $H(X) - \epsilon \leq -\frac{1}{n} \log p(x_1, x_2, \ldots, x_n) \leq H(X) + \epsilon$.
2. $Pr\{A^{(n)\}_\epsilon} > 1 - \epsilon$ for n sufficiently large.
3. $|A_\epsilon^{(n)}| \leq 2^{n(H(X)+\epsilon}$, where $|A|$ denotes the number of elements of the set A.
4. $|A_\epsilon^{(n)}| \geq (1-\epsilon)2^{n(H(X)-\epsilon)}$ for n sufficiently large.

Giving an interpretation of these properties we can say that property (1) collocates the probabilities of the elements of the typical set all close together (and also close to the entropy); property (2) states the high probability of the typical set, meaning that an encountered sample sequence will be very probably in the typical set; the properties (3) and (4) tell us that the number of elements composing the typical set is nearly 2^{nH}.

Proof. We will prove the properties one at a time:

1. The proof of property (1) is immediate from the definition of $A_\epsilon^{(n)}$. It is sufficient to take the logarithm of both sides of the inequation and divide by $-n$.

2. The second property follows from the AEP theorem, since the probability of the event $(X_1, X_2, \ldots, X_n) \in A_\epsilon^{(n)}$ tends to 1 as $n \to \infty$.
   Thus, for any $\delta > 0$, there exists an n_0 such that for all $n \geq n_0$ we have

   $$Pr\{|-\frac{1}{n}\log p(X_1, X_2, \ldots, X_n) - H(X)| < \epsilon\} > 1 - \delta$$

   Setting $\delta = \epsilon$ we obtain the second part of the theorem. We will use again the identification $\delta = \epsilon$ to simplify notation.

3. To prove property (3) we write

$$\begin{aligned} 1 &= \sum_{x \in \chi^n} p(x) \\ &\geq \sum_{x \in A_\epsilon^{(n)}} p(x) \\ &\geq \sum_{x \in A_\epsilon^{(n)}} 2^{-n(H(X)+\epsilon)} \\ &= 2^{-n(H(X)+\epsilon)} |A_\epsilon^{(n)}| \end{aligned}$$

where the second inequality follows from the definition of the typical set. Hence

$$2^{-n(H(X)+\epsilon)} \geq |A_\epsilon^{(n)}|$$

4. Finally, for sufficiently large n, $Pr\{A_\epsilon^{(n)}\} > 1 - \epsilon$, so that

$$\begin{aligned} 1 - \epsilon &< Pr\{A_\epsilon^{(n)}\} \\ &\leq \sum_{x \in A_\epsilon^{(n)} 2^{-n(H(X)-\epsilon)}} \\ &= 2^{-n(H(X)-\epsilon)} |A_\epsilon^{(n)}| \end{aligned}$$

where the second inequality follows from the definition of the typical set. Hence

$$(1 - \epsilon) 2^{-n(H(X)-\epsilon)} \leq |A_\epsilon^{(n)}|$$

which completes the proof for the properties of the typical set.

□

After all the theory has been exposed we can now apply the AEP theorem to construct a data compressor algorithm, able to reduce sequences to a size arbitrary close to the entropy, for sufficiently large files.

Let $X_1, X_2, \ldots, X_n$ be indipendent, identically distributed random variables, drawn according to the probability mass function p(x). We can image these sequences as takes from an input file we want to compress.

We wish to find short descriptions for such sequences, in order to reduce the size of the file without losing information.

We decide to apply the concept of typical set, dividing all the sequences in χ^n (or all the sequences present in our file, which are probably fewer than $|\chi^n|$) into two sets: the typical set $A_\epsilon^{(n)}$ and its complement.

The next step consists in ordering all elements in each set according to some order, for example lexicographic order. Now we are able to represent a sequence in $A_\epsilon^{(n)}$ using a short description: the position (i.e. the index) of the sequence in the set.

Since there are $\leq 2^{n(H+\epsilon)}$ sequences in $A_\epsilon^{(n)}$, the indexing requires no more than $n(H+\epsilon)+1$ bits. [The extra bit may be necessary because $n(H+\epsilon)$ may not be an integer.]

Similarly, we can index each sequence in the non-typical set by using not more than $n \log |\chi| + 1$ bits.

To distinguish between typical and non-typical sequences we prefix them by a single bit: this bit is set to 0 for typical sequences, to 1 for non-typical ones.

We have obtained a one-to-one code for all the sequence in χ^n, that is easy to decode. It is efficient? Yes, according to the next theorem.

Theorem (The Shannon's AEP compressor reaches entropy)**.** Let X^n be i.i.d. with probability mass function $p(x)$. Let $\epsilon > 0$. Then the Shannon's AEP compressor, described above, maps sequences x^n of length n into binary strings, such that the mapping is one-to-one (and therefore invertible) and

$$E[\frac{1}{n} l(X^n)] \leq H(X) + \epsilon$$

for n sufficiently large.

Proof. We denote x^n a sequence $x_1, x_2, \ldots, x_n$. Let $l(x^n)$ be the length of the codeword corresponding to x^n. If n is sufficiently large so that $Pr A_\epsilon^{(n)} \geq 1 - \epsilon$, the expected length of the codeword is

$$\begin{aligned}
E(l(X^n)) &= \sum_{x^n} p(x^n) l(x^n) \\
&= \sum_{x^n \in A_\epsilon^{(n)}} p(x^n) l(x^n) + \sum_{x^n \in A_\epsilon^{(n)c}} p(x^n) l(x^n) \\
&\leq \sum_{x^n \in A_\epsilon^{(n)}} p(x^n)(n(H+\epsilon)+2) \\
&\quad + \sum_{x^n \in A_\epsilon^{(n)c}} p(x^n)(n \log |\chi| + 2) \\
&= Pr A_\epsilon^{(n)} (n(H+\epsilon)+2) + Pr A_\epsilon^{(n)c} (n \log |\chi| + 2) \\
&\leq (1-\epsilon)(n(H+\epsilon)+2) + \epsilon(n \log |\chi| + 2) \\
&\leq n(H+\epsilon) + 2 + \epsilon n \log |\chi| \qquad\qquad = n(H+\epsilon')
\end{aligned}$$

where $\epsilon' = \epsilon + \epsilon \log |\chi| + \frac{2}{n}$ can be made arbitrarily small by an appropriate choice of ϵ followed by an appropriate choice of n.

□

Thus, we can represent sequences X^n using $nH(X)$ bis in the average.

Implementation of an AEP data compressor

The objective of this document is implement a compressor based on the AEP, but that is also efficient is real life scenarios.

Usually, one do not want to compress an infinite length string of symbol generated by an i.i.d. source, instead he want to compress a finite length segment of the stream (i.e. a file).

So, instead of considering all possible sequences an i.i.d source can generate, it is more efficient to consider only the sequences that appears in the file to compress.

In addition, a file has usually a lot less entropy that a pure string segment generated by an i.i.d. source. Considering only sequences appearing in the file we can take advantage of this fact.

This is the pragmatic point of view, we also include a brief consideration of an approach that follows the AEP theorem more closely in Appendix B. We won't use any fancy combinatorics in this section, only basic techniques exposed in an undergraduate course on algorithms and data structures.

The basic block for our algorithm will be the sequence, using symbols only as an intermediate element. We also assume that the length n of such sequences is fixed and provided as a parameter to the algorithm. We assume the length n is in bits.

We can decompose the problem in various subproblems:

1. First of all the compressor need to know what sequences are present in the file and the frequency of each sequence.
2. Using the list of frequencies the compressor must decide the most convenient size of the typical set. Note that we do not assume that this size is decided beforehand, and we don't use a parameter ϵ.
3. The compressor has now the information he needs to write the compressed file: it must first serialize the typical set in the file header, then encode each sequence of the file considering if the sequence is typical or not.

Each of these stages presents its own difficulties and tradeoffs.

Calculate the Frequency of Sequences

Suppose we want to divide a file of size N bytes into sequences of fixed length n bits, and calculate how many repetitions of each sequence are present in the file (i.e. the probability of these sequences in the file).

Having no prior information on the distribution of sequences, we can only assume a lower bound to the complexity of the algorithm that performs this operation: since to enumerate all the sequences the algorithm must read the file from the beginning to the end, its complexity is at least $\Omega(\frac{N}{n}) = \Omega(N)$.

Suppose now the algorithm can use a primitive to read a sequence at a time from the file, in order of appeareance of sequences in the file. It must keep track of what sequences are present, and count their frequencies.

To keep track of sequences, in the majority of cases, it is not possible to use a simple array a of length 2^n: even for sequences of a relative short length of 32 bits, the total size required by the array is 2^{32} bits = 4 Gigabytes.

A natural alternative to plain arrays is to use an hash map. But using an hash map for this type of application poses a few problems we want to avoid.

To remain efficient an hash map should have a low load factor of circa $fracnm$, where n is the number of elements inserted in the hash map, and m is the number of unique "cells" the hash map provides. If more than m elements are inserted in the hash map, then at least one cell contains more than one element. If too many elements are inserted the performance of the hash map degrades greatly, approaching linearity.

A solution to this problem consists in keeping track of the load factor and, when it increases more than a threshold, resizing the hash map to a greater value of m (usually m is doubled). But this operation can be very expensive, expecially if used in an compressor.

For this reason we preferred to use a data structure that has a very predictable behaviour and that consumes a memory "strictly" proportional to the number of elements it contains.

We opted for a balanced binary tree. While reading the file, this structure keeps track of the sequences encountered so far, and is able to tell us if a sequence has been already seen in $O(m)$ time, where m is the number of unique sequences in the tree.

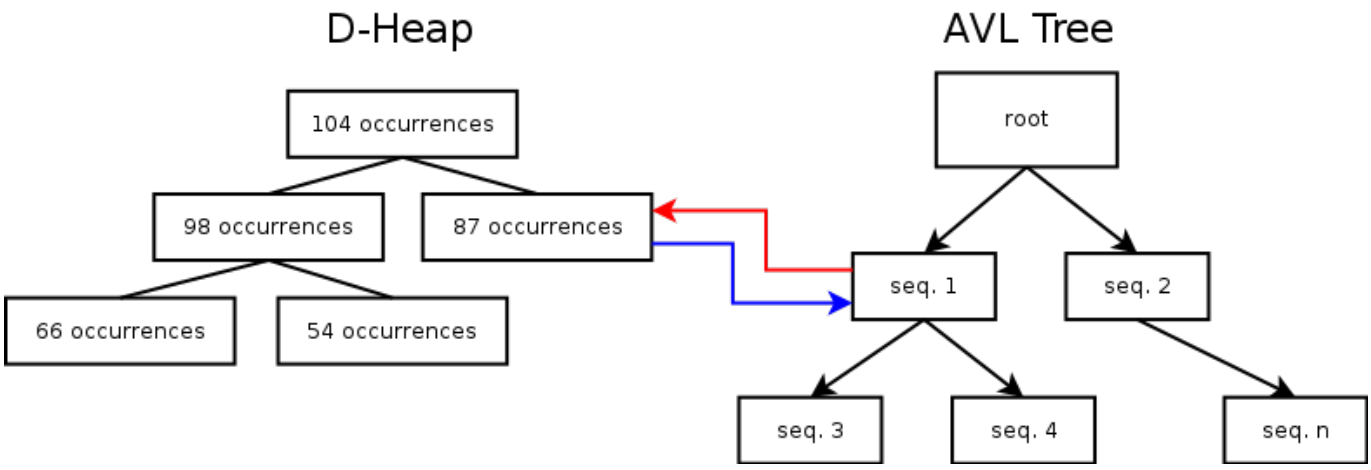
Now we should decide how to count occurrences of sequences. The simplest solution is adding a counter to each node of the tree, incremented after a sequence has been encountered in the file.

However, this solution forces us to run a sorting algorithm (e.g quicksort) on the tree each time we need the list of sequences in decreasing order of occurrences. This is not a problem if we only run the sorting algorithm after the file has been completely read.

It may be useful, however, to have this information ready during all the file-reading phase. We have achieved this result combining two different data structures: an AVL Tree and a D-Heap.

A D-Heap can be used both as a very simple type of priority queue and as the core component of the heap-sort algorithm. It is a data structure used to maintain the minimun or, as in our case, the maximum of a total ordered set of keys. The cost of finding the minimum (or maximum) element is constant, and inserting or increasing the "priority" of an element already present in the heap has logarithmic cost in the size of the heap.

We combine the two data structures in the following way: each node of the tree contains a copy of a binary sequence encountered in the file (used as the node key), and a pointer to an element of the D-Heap. The referenced element of the D-Heap contains a key equals to the number of occurrences of the sequence encoutered so far in the file, and a pointer back to the node of the tree.



Populating the D-Heap and incrementing its elements' keys while reading the input file is very similar to running an heapsort on that file. The main difference is that only unique sequences are kept.

An advantage of this approach is that, in any moment, we can extract the list of the k most common sequences from the heap. This operation can be implemented as a simple breadth-first search of the D-Heap, visiting only a few levels at the top of the tree.

This possibility will not be immediately exploited in the compressor, but will be used later in the document to improve the performance of the algorithm on files containing non homogeneous data sections.

Having described all the ingredients necessary for delineating a possible algorithm, we can delineate it in pseudo code (algorithm 1).

Input sequence length n in bits, uncompressed file F.
Output compressed file C.

Procedure:

reading phase:
while *not at the end of* F **do**
    read a sequence S of length n bits from F;
    if S *is in the Tree* **then**
        increment key of S in D-Heap;
    else
        insert S in Tree;
        insert S in D-Heap with key 1;
    end
end

determine the size of typical set:
extract from the D-Heap the list of sequences L, ordered by occurrences;
be m the number of unique sequences;
initialize *minsize* to the size of F;
count := 0;
for $i := 1$ **to** m **do**
    size := the calculated size of C using the first i sequences in the typical set;
    if $size < minsize$ **then**
        $minsize := size$;
        $count := i$;
    end
end

writing phase:
write compressed file header to C;
write list of *count* most common sequences to C;
reset F to the beginning;
while *not at the end of* F **do**
    read a sequence S of length n bits from F;
    if S *is in the typical set* **then**
        write bit 0 to C;
        write position of S in the typical list to C;
    else
        write bit 1 to C;
        write S to C;
    end
end

close files and cleanup;

Algorithm 1: AEP data compressor

Explanation of the phases of the algorithm

The reading phase uses the data structure described in the previous paragraph to store all unique sequences found in the file.

During this phase the file is completely read, a sequence at a time, and the number of occurrences of each unique sequence is calculated. When end-of-file is reached, every sequence can be looked up in the balanced tree and its frequency retrieved from the d-heap.

The second phase proceeds to assemble the typical set from data collected in the first phase: sequences are extracted from the d-heap in decreasing order of probability, and then the algorithm tries several sizes for the typical set to find the optimal one.

The formula used to compute the final size of the compressed file is the following:

$$
\begin{aligned}
S_{HeaderTypicalSet} &= C_{TypicalSet} * \left\lceil \log_2(C_{TypicalSet}) \right\rceil - 1 \\
S_{CompressedTypicalSequences} &= C_{TypicalSequencesInFile} * \left\lceil \log_2(C_{TypicalSet}) \right\rceil + 1 \\
S_{CompressedNontypicalSequences} &= (C_{TotSequencesInFile} - C_{TypicalSequencesInFile}) * (S_{Sequence} + 1)
\end{aligned}
$$

$$
\begin{aligned}
S_{FinalCompressedFile} &= S_{HeaderFixedFields} + S_{HeaderTypicalSet} \\
&\quad + S_{CompressedTypicalSequences} + S_{CompressedNontypicalSequences}
\end{aligned}
$$

where S variables indicate sizes in bits and C variables indicate a count of elements.

We note that an increase in the number of sequences in the typical set, corresponds in an increase of the number of bits required to represent the position of sequences in the set. For example, a typical set containing only two sequences, requires only a bit to distinguish between them (plus a suffix bit to mark the sequence as typical in the compressed file). If we increase the set to three sequences, we now need two bits to represent their position in the set.

For this reason, in the implementation, instead of adding a single sequence in each round, we double the typical set size, so that it always contains a number of sequences that is a power of two, avoiding wasted bits.

Once the optimal size for the typical is determined, the third phase writes the compressed file headers and begin to compress the sequences of the original file.

This is done rewinding the original file, and reading it again a sequence at a time. If the encountered sequence is typical, it will be encoded as its position in the typical set prefixed by a zero, otherwise it will be left unchanged but prefixed by a one.

The structure of the compressed file is the following: first some fixed header fields are present (to specify sequence count and length, and other parameters), followed by the serialized typical set. The rest of the file consists of encoded sequences.

To decompress the file, it is sufficient to read the typical set in a table, and decode the compressed sequences accordingly, using the table as a reference.

Benchmarks

To test the compression levels we can reach, we benchmarked an implementation of our algorithm against the well known tools gzip and bzip2 (gzip implements a variant of LZ-77, and bzip2 uses Burrows–Wheeler transform).

Both tools have been executed using a compression of -9, the best possible compression.

The machine used to conduct the test is a Thinkpad T500, with an amd64 Linux kernel 4.2.6-64. The processor used is Intel(R) Core(TM)2 Duo CPU T9600 2.80GHz, with 4GB DDR3 1066 MHz memory.

Calgary and Canterbury Corpora

The Calgary corpus was developed in the late 1980s, and during the 1990s became something of a de facto standard for lossless compression evaluation.

This corpus is composed by several small files, of various kind. We do not report the execution times of the programs since they are negligible.

Table 1: Calgary corpus results

File	**Description**	**Orig. Size**	**BZ2**	**GZ**	**SHAN (16)**
bib	bibliography	111261 (109K)	27467 (27K)	34900 (35K)	73788 (73K)
book1	fiction book	768771 (751K)	232598 (228K)	312281 (305K)	480076 (469K)
book2	non-fiction book	610856 (597K)	157443 (154K)	206158 (202K)	398028 (389K)
geo	geophysical data	102400 (100K)	56921 (56K)	68414 (67K)	73764 (73K)
news	USENET batch	377109 (369K)	118600 (116K)	144400 (142K)	262028 (256K)
obj1	object code VAX	21504 (21K)	10787 (11K)	10320 (11K)	17180 (17K)
obj2	object code Mac	246814 (242K)	76441 (75K)	81087 (80K)	174524 (171K)
paper1	technical paper	53161 (52K)	16558 (17K)	18543 (19K)	35684 (35K)
paper2	technical paper	82199 (81K)	82199 (25K)	29667 (29K)	51540 (51K)
pic	fax picture	513216 (502K)	49759 (49K)	52381 (52K)	137340 (135K)
progc	source C	39611 (39K)	39611 (13K)	13261 (13K)	27484 (27K)
progl	source LISP	71646 (70K)	71646 (16K)	16164 (16K)	45436 (45K)
progp	source PASCAL	49379 (49K)	10710 (11K)	11186 (11K)	33036 (33K)
trans	terminal session	93695 (92K)	17899 (18K)	18862 (19K)	64948 (64K)

The Canterbury corpus was developed in 1997 as an improved version of the Calgary corpus. The files were chosen because their results on existing compression algorithms are "typical".

As for the Calgary corpus, we do not report the execution times of the programs since they are negligible.

Table 2: Canterbury corpus results

File	Description	Orig. Size	BZ2	GZ	SHAN (16)
alice29	english txt	152089 (149K)	43202 (43K)	54191 (53K)	94676 (93K)
asyoulik	shakespeare	125179 (123K)	39569 (39K)	48829 (48K)	79908 (79K)
cp.html	HTML source	24603 (25K)	7624 (7.5K)	7981 (7.8K)	16988 (17K)
fields.c	C source	11150 (11K)	3039 (3.0K)	3136 (3.1K)	7500 (7.4K)
grammar.lsp	LISP source	3721 (3.7K)	1283 (1.3K)	1246 (1.3K)	2524 (2.5K)
kennedy.xls	excel	1029744 (1006K)	130280 (128K)	209733 (205K)	569532 (557K)
lcet10	technical txt	426754 (417K)	107706 (106K)	144429 (142K)	267316 (262K)
plrabn12	poetry	481861 (471K)	145577 (143K)	194277 (190K)	293852 (287K)
ptt5	CCITT test	513216 (502K)	49759 (49K)	52382 (52K)	137340 (135K)
sum	SPARC	38240 (38K)	12909 (13K)	12772 (13K)	27836 (28K)
xargs.1	man page	4227 (4.2K)	1762 (1.8K)	1756 (1.8K)	3116 (3.1K)

For all the executions we set the sequence size to 16 bits, a size that works well for short textual files.

For small files (under 200k) we were able to reduce the original size of a 30%/40%. However we could not reach the compression levels of gzip and bzip2 for files so small.

Results were better for medium sized files of the Canterbury corpus (e.g. kennedy.xls), where we almost halved file size. But even for medium sized files the compression levels of gzip and bzip2 are far better.

Pizza Chili Corpus

This corpus is the most recent.

It is composed of larger texts, from various application domains. We selected in particular three files, from three different collections, since they highlight peculiar differences in compressions.

We choose a size of 200 MB for each sample text.

The first sample is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions. Downloaded on June 9, 2005.

Table 3: Pizza Chili - Source results

File	source	
Description	JAVA and C source code	
Orig. size	209715200 (200MB)	
BZ2	39130584 (38MB)	0m29.606s
GZ	46928118 (45MB)	0m23.730s
SHAN (16)	150712132 (144MB)	2m44.878s
SHAN (24)	145229980 (139MB)	2m36.736s
SHAN (32)	126042260 (121MB)	2m16.152s
SHAN (40)	139680292 (134MB)	3m17.922s
SHAN (48)	147213100 (141MB)	3m14.430s
SHAN (56)	179276108 (171MB)	3m28.514s
SHAN (64)	126605620 (121MB)	2m15.771s

On the source sample the best results are obtained using longer sequences (32 or 64 bits). We are still far behind gzip and bzip2, but the compression level is better with large files.

The second sample is a concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. The headers have been removed to leave just the real text. Downloaded on May 4, 2005.

Table 4: Pizza Chili - English results

File	english	
Description	english text files	
Orig. size	209715200 (200MB)	
BZ2	58866860 (57M)	0m27.279s
GZ	78941298 (76MB)	0m29.215s
SHAN (16)	131026212 (125MB)	2m38.414s
SHAN (24)	128032148 (123MB)	2m24.619s
SHAN (32)	111702052 (107MB)	2m0.804s
SHAN (40)	124522892 (119MB)	2m53.432s
SHAN (48)	129347004 (124MB)	3m9.914s
SHAN (56)	172031980 (165MB)	3m36.191s
SHAN (64)	120328340 (115MB)	2m27.181s

On the english sample we almost reach a 50% compression level, and we are very close to the maximum compression level of gzip using sequences of length 32 bits.

The last sample is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project. Each of the 4 bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special characters. Downloaded on June 9, 2005.

Table 5: Pizza Chili - DNA results

File	dna	
Description	dna sequences	
Orig. size	209715200 (200MB)	
BZ2	54412336 (52MB)	0m28.440s
GZ	56671384 (55MB)	4m11.023s
SHAN (16)	65541132 (63M)	1m46.541s
SHAN (24)	61174388 (59M)	1m23.195s
SHAN (32)	58992620 (57M)	1m13.828s
SHAN (40)	57688188 (56M)	1m8.747s
SHAN (48)	56835884 (55M)	1m5.823s
SHAN (56)	56303716 (54M)	1m3.789s
SHAN (64)	56243348 (54M)	1m3.759s

The results of the DNA sample shows something peculiar: we were able to provide, using sequences of length 64 bits, a better compression level than gzip, and in a fourth of the time.

The DNA sample is somewhat peculiar, since it uses a very limited set of characters, and the encoding format of the original file (ascii text) takes a lot more space than what strictly necessary.

It also allows our algorithm to overcome its main handicap: sequences are expected at specific offsets in the file, and shifted sequences are difficult to handle. Gzip and bzip2 do not have this limitation and generally perform better on generic texts. But this last sample privileged the simplicity of our algorithm, and allowed it to work at its maximum efficiency.

We discuss some ways to overcome the limitations of our algorithm on general texts in the conlusion.

For each Pizza Chili sample analysed in this section we provide a table that illustrate the tradeoffs our algorithm computed to select the optimal size for the typical set. We only include the tables calculated for 32 bits sequences since is a very good sequence length for the samples and it allows us to compare the typical sets of the various samples.

The optimal typical set for the sample is in bold.

Table 6: Pizza Chili - Source - Typical set computations for 32 bits sequences.

Typical set size (over unique sequences)	**Typical sequences in file (over total sequences)**	**Typical set probability**
2 / 1114490	1539156 / 52428800	0.02935707092285156
4 / 1114490	1965691 / 52428800	0.03749258041381836
8 / 1114490	2515595 / 52428800	0.04798116683959961
16 / 1114490	3362186 / 52428800	0.06412860870361328
32 / 1114490	4540999 / 52428800	0.08661268234252929
64 / 1114490	6136711 / 52428800	0.1170484733581543
128 / 1114490	11592295 / 52428800	0.2211054801940918
256 / 1114490	10433066 / 52428800	0.1989949417114258
512 / 1114490	13474222 / 52428800	0.2570003890991211
1024 / 1114490	17224346 / 52428800	0.3285283279418945
2048 / 1114490	21737399 / 52428800	0.41460798263549803
4096 / 1114490	26748928 / 52428800	0.5101953125
8192 / 1114490	31915974 / 52428800	0.608748893737793
16384 / 1114490	36885333 / 52428800	0.7035318946838379
32768 / 1114490	41411338 / 52428800	0.7898585891723633
65536 / 1114490	**45244349 / 52428800**	**0.862967472076416**
131072 / 1114490	48192516 / 52428800	0.9191992950439453
262144 / 1114490	50279752 / 52428800	0.9590101623535157
524288 / 1114490	51616438 / 52428800	0.9845054244995117
1048576 / 1114490	52362886 / 52428800	0.9987427902221679

We note that for this semple the typical set is too large and has a low probability. This determined a low compression level.

Table 7: Pizza Chili - English - Typical set computations for 32 bits sequences.

Typical set size (over unique sequences)	**Typical sequences in file (over total sequences)**	**Typical set probability**
2 / 382398	1166056 / 52428800	0.022240753173828124
4 / 382398	1754873 / 52428800	0.0334715461730957
8 / 382398	2578972 / 52428800	0.04918998718261719
16 / 382398	3507375 / 52428800	0.06689786911010742
32 / 382398	4874203 / 52428800	0.09296804428100586
64 / 382398	6689655 / 52428800	0.12759504318237305
128 / 382398	9167514 / 52428800	0.17485645294189453
256 / 382398	12429222 / 52428800	0.23706859588623047
512 / 382398	16540655 / 52428800	0.3154879570007324
1024 / 382398	21667590 / 52428800	0.41327648162841796
2048 / 382398	27684245 / 52428800	0.5280350685119629
4096 / 382398	34323578 / 52428800	0.6546702957153321
8192 / 382398	40805271 / 52428800	0.7782987785339356
16384 / 382398	45958919 / 52428800	0.8765968132019043
32768 / 382398	**49269571 / 52428800**	**0.939742488861084**
65536 / 382398	51070195 / 52428800	0.9740866661071778
131072 / 382398	51929509 / 52428800	0.9904767799377442
262144 / 382398	52308546 / 52428800	0.9977063369750977

This sample is better: the typical set is smaller and has a better probability. But still not good enough to compete with gzip and bzip.

Table 8: Pizza Chili - DNA - Typical set computations for 32 bits sequences.

Typical set size (over unique sequences)	**Typical sequences in file (over total sequences)**	**Typical set probability**
2 / 1259	1623305 / 52428800	0.030962085723876952
4 / 1259	2697275 / 52428800	0.05144643783569336
8 / 1259	4406760 / 52428800	0.08405227661132812
16 / 1259	7551958 / 52428800	0.14404216766357422
32 / 1259	12978640 / 52428800	0.24754791259765624
64 / 1259	22067473 / 52428800	0.4209036445617676
128 / 1259	37187695 / 52428800	0.7092989921569824
256 / 1259	**52425739 / 52428800**	**0.9999416160583496**
512 / 1259	52427850 / 52428800	0.9999818801879883
1024 / 1259	52428565 / 52428800	0.9999955177307129

The compression level of the DNA sample was very good because the algorithm found a very small typical set (only 256 sequences) with a probability of over 99%. Almost every sequence of the file was reduced from an original 32 bits length to a $1 + \log_2(256) = 9$ bits.

Conclusions and possible improvements

We shown in this document that is relatively easy to implement a compressor that follows very closely the AEP theorem and is efficient enough given its simplicity.

In fact we only used well known techniques, available to every student that just attended an undergraduate course in data structure and algorithms.

The main handicap of this algorithm lies in it being able to detect typical sequences only at fixed offsets of the file, failing to recognize shifted sequences.

A possible way to overcome this limitation is allowing for variable length sequences in the typical set. After the invention of suffix trees by Weiner in 1973 [3], almost all subsequent compression algorithms exploited this data structure to index all variable length strings in the text to compress, reaching very good compresion level (e.g. LZ-77 relies heavily on suffix trees).

Another possibility is to maintain fixed length sequences but adopting some method to permute the text and make it easier to compress. This method is used by bzip2, that uses Burrows–Wheeler transforms [4] on the text to compress.

The running speed of the current program is acceptable, but can be greatly improved optimizing core section of the code. We have not tried to optimize it, relying only on the computational complexity to assure a polinomial execution time.

Examining the benchmarks we note that the execution speed is directly correlated to the size of the typical set. A great speed improvement can thus be obtained optimizing the search procedure that determines if a sequence is typical or not.

Also, finding a way to assemble a better typical set for generic texts automatically improve execution speed.

Bibliography

References

[1] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[2] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[3] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[5] Tadao Takaoka. An o(1) time algorithm for generating multiset permutations. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, ISAAC '99, pages 237–246, London, UK, UK, 1999. Springer-Verlag.

[6] Aaron Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 987–996, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[7] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition, 2011.

Appendix A: Compressor source code

Here we include the code of the compressor, and the header files and libraries developed for it.

shancomp.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <assert.h>

#include "avl/avl.h"
#include "2heap/2heap.h"
#include "bit_io/bit_io.h"

/* program name */
char *prog_name;

/*
  Header of compressed file.

  Also used as options structure
  for a program run.
*/
typedef struct {
    uint8_t  magic_byte;     // magic byte 'S'
    uint8_t  sequence_size;  // size of sequences in bits
    uint16_t typical_count;  // number of sequences in typical set
    uint32_t sequence_count; // number of sequences in compressed file
    uint32_t orig_size;      // original file size in bytes
} header_t;

/*
  Structure describing a sequence.

  Memorized in the  AVL Tree,
  to keep track of sequences
  in the DHeap.
```

```
*/
typedef struct {
    uint64_t sequence;
    dheap_node_t *ptr;
    uint32_t pos;
    uint32_t occurrences;
} sequence_t;

/*
  Allocate and initialize new sequence_t
*/
sequence_t *
sequence_new()
{
    sequence_t *seq = calloc(1, sizeof(*seq));

    if(seq == NULL) {
fprintf(stderr, "%s: calloc() failed.\n", prog_name);
exit(1);
    }

    seq->pos = -1;

    return seq;
}

/*
  Compare function for the type sequence_t
*/
int
sequence_cmp (void *a, void *b)
{
    sequence_t *sa = a;
    sequence_t *sb = b;

    assert(a != NULL);
    assert(b != NULL);

    return
      sa->sequence < sb->sequence ? -1 :
      sa->sequence > sb->sequence ?  1 : 0;
}

/*
 * Print bits
 */
void
print_bits (uint8_t *buf, uint32_t count)
```

```
{
    uint8_t mask = 128;

    while(count > 0) {
printf("%c", *buf & mask ? '1' : '0');

mask >>= 1;

if(mask == 0) {
    mask = 128;
    buf++;
}

count--;
    }
}

/*
  Return the highest set bit.

  Integer rounded base 2 logarithm.
*/
uint32_t
highest_bit(uint64_t n) {
    if (!n)
     return 0;

    int ret = 1;

    while (n >>= 1)
     ret++;

    return ret;
}

/*
  Hexadecimal dump of a buffer.
  Taken from: http://c2.com/cgi/wiki?HexDumpInManyProgrammingLanguages

  This entry contains certain functionality the others may not.

  The stop-start span does not need to be equal to width
  The command-line is checked for missing arguments
*/
void
hexdump(uint8_t *buffer, uint32_t index, uint32_t width)
{
    uint32_t i;
```

```
    uint32_t spacer;

    for (i = 0; i < index; i++) {
printf("%02x ",buffer[i]);
    }

    for (spacer = index; spacer < width; spacer++)
printf(" ");

    printf("| ");

    for (i = 0; i < index; i++) {
if (isprint(buffer[i])) printf("%c",buffer[i]);
else printf(".");
    }

    printf(" |\n");
}

/*
  Fill typical set and determine its optimum size
*/
void
fill_typical_set(node_t *tree, dheap_t *heap, uint32_t sequence_size,
 sequence_t *typical_set[], uint32_t *typical_count,
 uint32_t *sequence_count)
{
    uint32_t count = 0, best_count, max_count;
    uint64_t best_file_size = 0;

    uint32_t cumulative_occurrences = 0, tot_occurrences = 0;

    dheap_node_t *dnode;

    /* fill typical set */

    while((dnode = dheap_find_max(heap)) != NULL) {
        sequence_t seq, *seq_ptr;

uint32_t occurrences = dnode->key;

        seq.sequence = dheap_delete_max(heap);
        seq_ptr = avl_search_and_insert(tree, &seq, sequence_cmp);

seq_ptr->occurrences = occurrences;
        typical_set[count] = seq_ptr;
        seq_ptr->pos = count;
```

```
tot_occurrences += occurrences;
        count++;
    }

    max_count = count;

    /* calculate best typical set size */

    for(count = 0; count < max_count; count++) {

if(count > 1 && (count & (count - 1)) == 0) { // if count is a power of two

    uint64_t file_size =
// size of typical sequences list
(count * highest_bit(count) - 1) +
// size of typical sequences in compressed file
(cumulative_occurrences * highest_bit(count) + 1) +
// size of non-typical sequences in compressed file
((tot_occurrences - cumulative_occurrences) * (sequence_size + 1));

    if(best_file_size == 0 || file_size < best_file_size) {
best_count = count;
best_file_size = file_size;
    }
}

cumulative_occurrences += typical_set[count]->occurrences;
    }

    *typical_count = best_count;
    *sequence_count = tot_occurrences;
}

/*
  Read sequences from input file and
  fill tree and dheap.
*/
void
read_sequences(FILE *input, node_t *tree, dheap_t *heap,
       uint32_t sequence_size, uint32_t *sequence_count)
{
    sequence_t *seq, *new_seq;

    // TODO: check remaining bits on file!
    // (maybe add a function in bit_io to flush buffered bits...)

    *sequence_count = 0;
```

```
    new_seq = sequence_new(); // alloc memory to save a sequence

    while ((read_bits(input, &new_seq->sequence, sequence_size)) != 0) {

        seq = avl_search_and_insert(tree, new_seq, sequence_cmp);

        if(seq == NULL) {
            fprintf(stderr, "%s: avl_search_and_insert() failed\n", prog_name);
            exit(1);
        }

        if(seq->ptr == NULL) {

            /* new sequence */

            seq->ptr = dheap_insert(heap, seq->sequence, 1);

            if(seq->ptr == NULL) {
                fprintf(stderr, "%s: dheap_insert() failed\n", prog_name);
                exit(1);
            }

    new_seq = sequence_new(); // alloc a new sequence struct

    (*sequence_count)++;

        } else {

            /* sequence already seen */

            dheap_increase_key(heap, seq->ptr);

        }

    }

    free(new_seq); // since last allocated sequence_t is unused...
}

/*
  Write compressed sequences to output file.
*/
void
write_sequences(FILE *input, FILE *output, node_t *tree,
uint32_t sequence_size, uint32_t typical_count)
{
    uint64_t seq_data;
```

```
    fseek(input, 0, SEEK_SET);  // rewind the file
    read_bits(NULL, NULL, 0); // clear the read buffer

    int cnt = 0, cnt2 = 0;
    int bits;

    seq_data = 0;
    while ((bits = read_bits(input, &seq_data, sequence_size)) == sequence_size) {
        sequence_t seq, *seq_ptr;

        seq.sequence = seq_data;
        seq.pos = -1;
        seq_ptr = avl_search(tree, &seq, sequence_cmp);

        assert(seq_ptr != NULL);

        if(seq_ptr->pos < typical_count) {

            /* sequence in the typical set */

            cnt++;

    uint64_t seq_pos = seq_ptr->pos;
    seq_pos <<= (sizeof(seq_pos) * 8) - (highest_bit(typical_count) - 1);

            write_bits(output, 0, 1);
            write_bits(output, seq_pos, highest_bit(typical_count) - 1);
        }

        else {

            /* sequence not in the typical set */

            cnt2++;

            write_bits(output, 1ULL << 63, 1);
            write_bits(output, seq_data, sequence_size);
        }

seq_data = 0;
    }

    // trailing bits

    if(bits) {
write_bits(output, 1ULL << 63, 1);
write_bits(output, seq_data, sequence_size);
    }
```

```
    write_bits(output, 0, 0);

}

/*
  Read compressed file header
*/
void
read_header(FILE *input, header_t *header)
{
    // read magic byte

    if(fread(&header->magic_byte, 1, sizeof(header->magic_byte), input) <
       sizeof(header->magic_byte)) {
fprintf(stderr, "%s: compressed file too short.\n", prog_name);
exit(1);
    }

    if(header->magic_byte != 'S') {
fprintf(stderr, "%s: invalid compressed file.\n", prog_name);
exit(1);
    }

    // read sequence size

    if(fread(&header->sequence_size, 1, sizeof(header->sequence_size), input) <
       sizeof(header->sequence_size)) {
fprintf(stderr, "%s: compressed file too short.\n", prog_name);
exit(1);
    }

    if(header->sequence_size < 1 || header->sequence_size > 64) {
fprintf(stderr, "%s: sequence size out of range.\n", prog_name);
exit(1);
    }

    // read typical count
    if(fread(&header->typical_count, 1, sizeof(header->typical_count), input) <
       sizeof(header->typical_count)) {
fprintf(stderr, "%s: compressed file too short.\n", prog_name);
exit(1);
    }

    if(header->typical_count < 1) {
fprintf(stderr, "%s: typical set size out of range.\n", prog_name);
exit(1);
    }
```

```
    // read sequence count

    if(fread(&header->sequence_count, 1, sizeof(header->sequence_count), input) <
       sizeof(header->sequence_count)) {
fprintf(stderr, "%s: compressed file too short.\n", prog_name);
exit(1);
    }

    if(header->sequence_count < 1) {
fprintf(stderr, "%s: number of sequences out of range.\n", prog_name);
exit(1);
    }

    // read original file size

    if(fread(&header->orig_size, 1, sizeof(header->orig_size), input) <
       sizeof(header->orig_size)) {
fprintf(stderr, "%s: compressed file too short.\n", prog_name);
exit(1);
    }

    if(header->orig_size < 1) {
fprintf(stderr, "%s: original file size out of range.\n", prog_name);
exit(1);
    }

}

/*
  Main function (decompressor)
*/
int
main_decompress (int argc, char *argv[])
{
    FILE *input, *output;
    header_t header;

    uint32_t typical_sequence_size;
    uint64_t sequence;

    uint64_t *typical_set;

    int i;

    /*
      === Initialization ===
    */
```

```
    prog_name = argv[0];

    if(argc < 3) {
printf("usage: %s input_file output_file\n",
       prog_name);
return 1;
    }

    input = fopen(argv[1], "r");
    if(input == NULL) {
fprintf(stderr, "%s: %s\n", prog_name, strerror(errno));
return 1;
    }

    output = fopen(argv[2], "w");
    if(output == NULL) {
fprintf(stderr, "%s: %s\n", prog_name, strerror(errno));
return 1;
    }

    /*
      === Phase 1 ===
    */

    /* read file header */

    read_header(input, &header);

    typical_sequence_size = highest_bit(header.typical_count) - 1;

    /*
      === Phase 2 ===
    */

    typical_set = calloc(header.typical_count, sizeof(*typical_set));

    /* read typical set */

    for(i = 0; i < header.typical_count; i++) {

if(!read_bits(input, &typical_set[i], header.sequence_size)) {
    fprintf(stderr, "%s: input file too short.\n", prog_name);
    return 1;
}

    }
```

```
    /*
      === Phase 3 ===
     */

    /* read compressed words from input file
       and write them uncompressed to output file */

    while((read_bits(input, &sequence, 1)) != 0) {

if(sequence) {

    /* non typical sequence */

    if(!read_bits(input, &sequence, header.sequence_size)) {
fprintf(stderr, "%s: input file too short.\n", prog_name);
return 1;
    }

    write_bits(output, sequence, header.sequence_size);

} else {

    /* typical sequence */

    int pos;

    if(!read_bits(input, &sequence, typical_sequence_size)) {
fprintf(stderr, "%s: input file too short.\n", prog_name);
return 1;
    }

    pos = sequence >> ((sizeof(sequence) * 8) - typical_sequence_size);

    write_bits(output, typical_set[pos], header.sequence_size);

}

    }

    // trailing bits
    write_bits(output, 0, 0);

    /* adjust output file size */

    fflush(output);
    ftruncate(fileno(output), header.orig_size);

    /* cleanup */
```

```
    free(typical_set);

    fclose(output);
    fclose(input);

    return 0;
}

/*
  Main function (compressor)
*/
int
main (int argc, char *argv[])
{
    FILE *input, *output;
    struct stat st;

    uint32_t sequence_size, typical_count, count;
    uint64_t sequence;

    uint32_t unique_sequence_count, sequence_count;

    header_t header = {'S', 0, 0, 0};

    sequence_t **typical_set;

    node_t  *tree = NULL;
    dheap_t *heap = NULL;

    /*
      === Select compressor or decompressor ===
    */

    if(strstr(argv[0], "shandecomp")) {
return main_decompress(argc, argv);
    }

    /*
      === Initialization ===
    */

    prog_name = argv[0];

    if(argc < 4) {
printf("usage: %s input_file output_file sequence_size\n",
       prog_name);
return 1;
```

```
    }

    sequence_size = atoi(argv[3]);

    if(sequence_size < 1) {
        fprintf(stderr, "%s: sequence sizes smaller than 1-bit are not supported.\n",
prog_name);
        return 1;
    }

    if(sequence_size > 64) {
        fprintf(stderr, "%s: sequence sizes larger than 64-bits are not supported.\n",
prog_name);
        return 1;
    }

    // TODO: limit typical set size?

    if((heap = dheap_new(0)) == NULL) {
        fprintf(stderr, "%s: error allocating memory for dheap.\n", prog_name);
        return 1;
    }

    if((tree = avl_new()) == NULL) {
        fprintf(stderr, "%s: error allocating memory for avl tree.\n", prog_name);
        return 1;
    }

    input = fopen(argv[1], "r");
    if(input == NULL) {
fprintf(stderr, "%s: %s\n", prog_name, strerror(errno));
return 1;
    }

    fstat(fileno(input), &st);

    output = fopen(argv[2], "w");
    if(output == NULL) {
fprintf(stderr, "%s: %s\n", prog_name, strerror(errno));
return 1;
    }

    /*
      === Phase 1 ===
    */

    /* Read sequences and fill tree and dheap structures */
```

```
    read_sequences(input, tree, heap, sequence_size, &unique_sequence_count);

    /*
       Extract encountered sequences,
       in decreasing order of occurrence,
       and fill the typical set.

       Also determine the optimal typical set size.
    */

    if((typical_set = calloc(unique_sequence_count, sizeof(*typical_set))) == NULL) {
        fprintf(stderr, "%s: malloc() failed.\n", prog_name);
        return 1;
    }

    fill_typical_set(tree, heap, sequence_size, typical_set,
     &typical_count, &sequence_count);

    /*
      === Phase 2 ===
    */

    /* Write compressed file header */

    header.sequence_size  = sequence_size;
    header.typical_count  = typical_count;
    header.sequence_count = sequence_count;
    header.orig_size      = st.st_size;

    fwrite(&header, sizeof(header), 1, output);

    /* write typical set */

    for(count = 0; count < typical_count; count++) {
write_bits(output, typical_set[count]->sequence, sequence_size);
    }

    /* Write compressed file */

    write_sequences(input, output, tree, sequence_size, typical_count);

    /* cleanup */

    fclose(output);
    fclose(input);

    dheap_free(heap);
```

```
    // Note: no avl tree free

    free(typical_set);

    return 0;
}
```

bit_io/bit_io.h

```
/*
 * Functions for bit-sequences input/output.
 *
 * All the functions operate in a little-endian fashion,
 * to reduce code complexity.
 *
 * Emanuele Acri - crossbower@gmail.com - 2015
 */

#include <stdio.h>
#include <stdint.h>

#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

/*
 * Mask to obtain a bit sequence
 * of the given length.
 */
static uint64_t
bit_mask (int length)
{
    uint64_t mask = UINT64_C(0xFFFFFFFFFFFFFFFF);

    mask <<= (64 - length);

    return mask;
}

/*
 * Read a bit-sequence.
 * The max supported length is 64 (bits).
 *
 * To flush the buffer set the length to 0.
 *
 * Return the number of bits read on success, 0 on error.
 */
```

```
int8_t
read_bits (FILE *in, uint64_t *out, int length)
{
    static uint64_t buff = 0;
    static uint32_t buff_len = 0;

    uint32_t m = 0, n = 0;

    if(length == 0) {

        /* flush the buffer */

        buff = 0;
        buff_len = 0;

        return 1;
    }

    /* read already buffered part */

    n = MIN(buff_len, length);

    *out = buff & bit_mask(n);

    buff <<= n;

    buff_len -= n;
    length   -= n;

    if(length > 0) {

        /* read new part from file */

buff = 0;
        buff_len = fread(&buff, 1, 8, in);

if(buff_len) buff_len = 64;

buff = htobe64(buff);

m = MIN(buff_len, length);

*out |= buff >> n;
*out &= bit_mask(n + m);

        if(buff_len <= length) {
    buff = 0;
    buff_len = 0;
```

```
} else {
    buff <<= length;
    buff_len -= length;
}
    }

    return n + m;
}

/*
 * Write a bit-sequence.
 * The max supported length is 64 (bits).
 *
 * To flush the buffer set the length to 0.
 *
 * Return the number of bits written on success, 0 on error.
 */
int
write_bits (FILE *out, uint64_t in, int length)
{
    static uint64_t buff = 0x0;
    static int buff_len = 0;

    uint64_t mask;
    int n, m;

    if(length == 0) {

        /* flush the buffer */

fwrite(&buff, 1, sizeof(buff), out);

        buff = 0;
        buff_len = 0;

        return 1;
    }

    /* store the first part */

    m = MIN(64, buff_len + length);

    buff |= in >> buff_len;
    buff &= bit_mask(m);

    n = MIN(64 - buff_len, length);

    buff_len += n;
```

```
    length   -= n;

    if(buff_len == 64) {

        /* write if buffer is full */

buff = htobe64(buff);

        if (fwrite(&buff, 1, sizeof(buff), out) != sizeof(buff))
            return 0;

        buff = 0;
        buff_len = 0;
    }

    if(length > 0) {

        /* store the second part */

        buff = in << n;
        buff_len = length;
    }

    return n + length;
}
```

2heap/2heap.h

```
/*
 *   2Heap Priority Queue implementation,
 *   ordered by maximum key.
 *
 *   Emanuele Acri - crossbower@gmail.com - 2015
 */

#ifndef DHEAP_H
#define DHEAP_H

#include <stdint.h>

/*
  Node of the 2heap.
*/
typedef struct {
    uint64_t value; /* value of the node */
    uint32_t key;   /* key of the node */
```

```
    uint32_t pos;   /* position of the node in the heap array */
} dheap_node_t;

/*
  The 2heap structure.

  We use an array to store the heap.
*/
typedef struct {
    uint32_t size;        /* size of the array, in elements */
    uint32_t last;        /* last empty node, next inserted leaf position */
    dheap_node_t **array; /* array containing the heap */
} dheap_t;

/*
  Create a new 2heap.

  It is possible to specify an estimate
  of the number of needed node.
*/
dheap_t * dheap_new(uint32_t);

/*
  Delete a 2heap.

  A pointer to the 2heap must be provided.
*/
void dheap_free(dheap_t *);

/*
  Return the max node.

  A pointer to the 2heap must be provided.
*/
dheap_node_t * dheap_find_max(dheap_t *);

/*
  Insert a new node.

  A pointer to the 2heap must be provided.

  The second argument specify the value of the node,
  the last one the key of the node.

  A pointer to the node is returned.
*/
dheap_node_t * dheap_insert(dheap_t *, uint64_t, uint32_t);
```

```
/*
  Delete the maximum node.

  A pointer to the 2heap must be provided.

  Return the value of the maximum node.
*/
uint64_t dheap_delete_max(dheap_t *);

/*
  Increase the key of a node.

  A pointer to the 2heap must be provided.
  The second argument is a pointer to the node to increment.

  Return the new key of the node.
*/
uint32_t dheap_increase_key(dheap_t *, dheap_node_t *);

#endif /* DHEAP_H */
```

2heap/2heap.c

```
/*
 *   2Heap Priority Queue implementation.
 *
 *   Emanuele Acri - crossbower@gmail.com - 2015
 */

#include "2heap.h"

#include <stdlib.h>
#include <assert.h>

/*
  Swap two nodes of the heap.

  Auxiliary function.
*/
static void
swap (dheap_t *heap, uint32_t pos1, uint32_t pos2)
{
    dheap_node_t *tmp = heap->array[pos1];
    heap->array[pos1] = heap->array[pos2];
    heap->array[pos2] = tmp;
```

```
    heap->array[pos1]->pos = pos1;
    heap->array[pos2]->pos = pos2;
}

/*
  Move up a node in the heap.

  Auxiliary function.
*/
static void
move_up (dheap_t *heap, uint32_t pos)
{
    assert(pos >= 1);

    while(pos != 1 &&
          heap->array[pos]->key > heap->array[pos/2]->key) {

        /* swap child and parent */
        swap(heap, pos, pos/2);

        pos /= 2;
    }
}

/*
  Move down a node in the heap.

  Auxiliary function.
*/
static void
move_down (dheap_t *heap, uint32_t pos)
{
    while(1) {
        uint32_t max_child_key;
        uint32_t max_child_num = 3;

        if((pos*2 < heap->last) && (heap->array[pos*2] != NULL)) {
            max_child_key = heap->array[pos*2]->key;
            max_child_num = 0;
        }

        if((pos*2+1 < heap->last) && (heap->array[pos*2+1] != NULL)) {
            if(max_child_key < heap->array[pos*2+1]->key) {
                max_child_key = heap->array[pos*2+1]->key;
                max_child_num = 1;
            }
        }
```

```
        if(max_child_num == 3) /* no children */
            break;

        if(max_child_key <= heap->array[pos]->key) /* correct position found */
            break;

        /* swap with maximum child */

        swap(heap, pos, pos*2+max_child_num);

        pos = pos*2+max_child_num;
    }
}

/*
  Create a new 2heap.

  It is possible to specify an estimate
  of the number of needed node.
*/
dheap_t *
dheap_new (uint32_t nodes)
{
    dheap_t * heap;

    if(nodes < 4)
        nodes = 4;

    heap = malloc(sizeof(*heap));
    assert(heap);

    heap->size = nodes;
    heap->last = 1; /* skip the first position */

    heap->array = calloc(nodes, sizeof(dheap_node_t *));
    assert(heap->array);

    return heap;
}

/*
  Delete a 2heap.

  A pointer to the 2heap must be provided.
*/
void
dheap_free (dheap_t *heap)
{
```

```
    assert(heap);
    assert(heap->array);

    free(heap->array);
    free(heap);
}

/*
  Return the max node.

  A pointer to the 2heap must be provided.
*/
dheap_node_t *
dheap_find_max (dheap_t *heap)
{
    assert(heap);

    if(heap->last == 1) /* heap empty */
        return NULL;

    return heap->array[1];
}

/*
  Insert a new node.

  A pointer to the 2heap must be provided.

  The second argument specify the value of the node,
  the last one the key of the node.

  A pointer to the node is returned.
*/
dheap_node_t *
dheap_insert (dheap_t *heap, uint64_t value, uint32_t key)
{
    assert(heap);

    dheap_node_t *node = malloc(sizeof(*node));
    assert(node);

    node->value = value;
    node->key = key;

    if(heap->size == heap->last) {
        /* we need more space to store the node */
        heap->size *= 2;
        heap->array = realloc(heap->array, heap->size * sizeof(dheap_node_t *));
```

```
        assert(heap->array);
    }

    heap->array[heap->last] = node;
    node->pos = heap->last;
    move_up(heap, heap->last);

    heap->last++;

    return node;
}

/*
  Delete the maximum node.

  A pointer to the 2heap must be provided.

  Return the value of the maximum node.
*/
uint64_t
dheap_delete_max (dheap_t *heap)
{
    assert(heap);
    assert(heap->array);

    /* save the root */

    dheap_node_t *max = dheap_find_max(heap);
    assert(max);

    /* move the last leaf in the radix */

    swap(heap, 1, heap->last - 1);

    heap->last--;

    move_down(heap, 1);

    /* delete the root */

    uint64_t value = max->value;
    free(max);

    return value;
}

/*
  Increase the key of a node.
```

```
  A pointer to the 2heap must be provided.
  The second argument is a pointer to the node to increment.

  Return the new key of the node.
*/
uint32_t
dheap_increase_key (dheap_t *heap, dheap_node_t *node)
{
    assert(heap);
    assert(heap->array);
    assert(node);

    node->key++;
    move_up(heap, node->pos);

    return node->key;
}
```

avl/avl.h

```
/*
  AVL Trees implementation.

  From Knuth III book, pages 458-475

  Emanuele Acri - crossbower@gmail.com - 2015
*/

#ifndef AVL_TREE_H
#define AVL_TREE_H

#include <stdint.h>

/*
  Node of a tree
*/
typedef struct node_t {
     /* navigation links */
     struct node_t *left, *right;

     /* balance factor */
     int8_t balance;

     /* data */
     void *data;
```

```
} node_t;

/*
  Create a new avl tree.

  This functions returns the special header node of the tree.

  This node has the real tree's root as its right child,
  and its balance field is used to keep track of the
  overall height of the tree.
*/
node_t *avl_new(void);

/*
  Create a new avl node.
*/
node_t *avl_new_node(void);

/*
  Test if the tree is empty
*/
int avl_empty(node_t *hdr);

/*
  Search.

  This function searches for the given value.
  If it is found, the node containing the searched
  value is returned.
 */
void *avl_search(node_t *hdr, void *searched, int (*cmp)(void *searched, void *data));

/*
  Search and insertion.

  This function searches for the given value.
  If it is not found, a new node is inserted in the tree.

  In both cases, the node containing the searched value is
  returned.
 */
void *avl_search_and_insert(node_t *hdr, void *searched, int (*cmp)(void *searched, void *da

#endif /* AVL_TREE_H */
```

avl/avl.c

```
/*
  AVL Trees implementation.

  From Knuth III book.

  Emanuele Acri - crossbower@gmail.com - 2015
*/

#include "avl.h"

#include <stdlib.h>
#include <assert.h>

/*
  Create a new avl tree.

  This functions returns the special header node of the tree.

  This node has the real tree's root as its right child,
  and its balance field is used to keep track of the
  overall height of the tree.
*/
node_t *
avl_new (void)
{
     return avl_new_node();
}

/*
  Create a new avl node.
*/
node_t *
avl_new_node (void)
{
     node_t *node;
     node = calloc(sizeof(*node), 1);
     return node;
}

/*
  Test if the tree is empty
*/
int
avl_empty (node_t *hdr)
{
```

```
     assert(hdr != NULL);

     return hdr->right == NULL && hdr->balance == 0;
}

/*
  Search.

  This function searches for the given value.

  If it is found, the node containing the searched
  value is returned. Otherwise NULL is returned.
 */
void *
avl_search (node_t *hdr, void *searched, int (*cmp)(void *searched, void *data))
{
     node_t *p;  /* will move down in the tree */
     node_t *q;
     int a;

     assert(hdr      != NULL);
     assert(searched != NULL);
     assert(cmp      != NULL);

     /* check empty tree */
     if(avl_empty(hdr)) {
         return NULL;
     }

     /* search phase */

     p = hdr->right;

     while(p != NULL) {

          a = cmp(searched, p->data);

          if(a == 0) {
               return p->data;
          }

          else if(a < 0) {
               p = p->left;
          }

          else {
               p = p->right;
          }
```

```
    }

    return NULL;
}


/*
  Search and insertion.

  This function searches for the given value.
  If it is not found, a new node is inserted in the tree.

  In both cases, the node containing the searched value is
  returned.

  After an insertion, the tree will be rebalanced, if needed.
 */
void *
avl_search_and_insert (node_t *hdr, void *searched, int (*cmp)(void *searched, void *data))
{
    node_t *p;  /* will move down in the tree */
    node_t *s;  /* will point to the node where rebalancing may be necessary */
    node_t *t;  /* will point to the parent of s */

    node_t *q, *r;

    int a;

    assert(hdr      != NULL);
    assert(searched != NULL);
    assert(cmp      != NULL);

    /* check empty tree */
    if(avl_empty(hdr)) {
        q = avl_new_node();

        hdr->right = q;
        hdr->balance++;

        q->data = searched;

        return q->data;
    }

    /* search phase */

    t = hdr;
```

```
s = p = hdr->right;

while(1) {

     a = cmp(searched, p->data);

     if(a == 0) {
          return p->data;
     }

     else if(a < 0) {

          q = p->left;

          if(q == NULL) {
               q = avl_new_node();
               p->left = q;

               break;
          }

     }

     else {

          q = p->right;

          if(q == NULL) {
               q = avl_new_node();
               p->right = q;

               break;
          }

     }

     if(q->balance != 0) {
          t = p;
          s = q;
     }

     p = q;
}

/*
  insertion phase:
  we have just linked a new node q to the tree
*/
```

```
q->data = searched;

/* adjust balance factors */

a = cmp(searched, s->data);

if(a < 0) r = p = s->left;
else      r = p = s->right;

while(p != q) {

     if(cmp(searched, p->data) < 0) {
          p->balance = -1;
          p = p->left;
     } else {
          p->balance = +1;
          p = p->right;
     }
}

if(s->balance == 0) {
     /* the tree has grown higher */

     s->balance = a;
     hdr->balance++;

     return q->data;
}

else if(s->balance == -a) {
     /* the tree has gotten more balanced */

     s->balance = 0;

     return q->data;
}

/* the tree has gotten out of balance ... */

/* rebalance phase */

if(r->balance == a) {
     /* single rotation */

     p = r;

     if(a == -1) {
```

```
          s->left  = r->right;
          r->right = s;
     } else {
          s->right = r->left;
          r->left  = s;
     }

     s->balance = r->balance = 0;
}

else {
     /* double rotation */

     if(a < 0) {

          p = r->right;

          r->right = p->left;
          p->left  = r;

          s->left  = p->right;
          p->right = s;

     } else {

          p = r->left;

          r->left  = p->right;
          p->right = r;

          s->right = p->left;
          p->left  = s;

     }

     if(p->balance == a) {
          s->balance = -a;
          r->balance =  0;
     } else if(p->balance == 0) {
          s->balance = 0;
          r->balance = 0;
     } else {
          s->balance = 0;
          r->balance = a;
     }

     p->balance = 0;
}
```

```
    /* final touch */

    if(s == t->right) t->right = p;
    else              t->left  = p;

    return q->data;
}
```

Appendix B: Some reflections on the combinatorics of the problem

This appendix describe a different approach to the problem, describing briefly how an algorithm, which more closely follows the AEP theorem, should behave.

Here we abandon the idea of exploiting a known file, to extract only the sequences that are actually present in it, but try to delineate the problem in a more general (and combinatoric way).

Non-binary alphabets

We know that the symbols of any alphabet χ can be encoded using binary strings, for all finite cardinalities $|\chi|$. Developing an algorithm that only deals with binary digits as symbols is simpler that the general case, that uses an arbitrary number of symbols.

This approach has however some disadvantages when encoding a file that is byte-oriented (and not bit-oriented) as is usually the case in modern computer architectures.

The first one is of pragmatic nature: since operating systems only offer primitives for reading and writing sequences of bytes, reading and writing single bits requires buffering and is generally slower. In addition the code that deals with bit I/O can be quite complex (we reduced the complexity of bit I/O in our implementation fixing a maximum length of 64 bits for sequences, see

`bit\_io/bit\_io.h`

).

We can not completely avoid bit I/O, since it is required to efficiently write a compressed file, but at least there should be the possibility to use using standard I/O primitives to read and write the original file we want to compress.

The second disadvantage is that, being the files we want to compress byte-oriented, if we subdivide the bytes composing these files into smaller parts, we lose information about admitted combinations of bits (i.e. valid and unvalid bytes), and generally obtain a worse compression level.

This is easy to see in the case of an ascii text file, where we may have an equal probability of observing a bit of value 0 or of value 1, but, of 256 possible sequences of bits (forming a byte), only about 60 of these are actually used.

For this reason, it would be better for our algorithm to be able to deal with alphabets composed by an arbitrary number of symbols (e.g. 256 symbols for an alphabet composed by all the possible values of a byte).

Some intuition on how this algorithm can operate is provided by the exercise number 3.13 of the

Thomas and Cover. The exercise ask to calculate a typical set for a given probability distribution, sequence size and ϵ. This involves the computation of a table of probabilities for the possible sequences (provided by the authors).

However the exercise carefully choose a binary alphabet for the computations, making them somewhat easier than in the general case.

In the following chapter we first describe a generalization of the same exercise, which will serve as a basis for the derivation of an algorithm for the task.

Calculation of the typical set for alphabets with cardinality greater that two

The problem we want to solve is the following: given a probability distribution $p_1, p_2, ..., p_m$ for an alphabet χ of $|\chi| = m$ symbols, and given a sequence length of n and a number ϵ, we want to enumerate all the sequences that fall in a typical set $A_\epsilon^{(n)}$.

The problem deals with probabilities of sequences, and this allows a simple trick. Consider an arbitrary sequence $s \in \chi^n$ having probability $p(s)$. A permutation s' of s will also have probability $p(s)$, since the product is commutative and the probability of a sequence is the product of the probabilities of its symbols.

To clarify the concept, consider the sequence $AABBC$ and probabilities for its symbols $p(A) = 0.5, p(B) = 0.3, p(C) = 0.2$.

The probability of the sequence is the product: $p(AABBC) = p(A)^2 * p(B)^2 * p(C)$.

The commutativity of the multiplication tell us that all the permutations of the sequences, for example:
ABABC
BBAAC
CBBAA
...
have all the same probability.

We are thus able to calculate the "shape" of the typical set before enumerating the single sequences composing it.

So, to fill our typical set we first try to find a "set" of symbols for the most probable sequence, and then expand it to all its possible permutations, filling a portion of the typical set. Next we find the second most probable "set" of symbols to use, and expand it...

We can continue this way until the tipical set reaches a probability of $1 - \epsilon$.

Generating all combinations of a multiset

The "sets" of symbols we alluded in the previous paragraphs are in fact called multisets.

A multiset is an extension of the concept of a set: while a set can contain only one occurrence of any given element, a multiset may contain multiple occurrences of the same element. Each element of a multiset is associated to its multiplicity in the multiset.

In choosing not only binary alphabets for our algorithm, we in fact entered into the realm of multisets.

Consider the following problem: given a set of m symbols, we want to find all the possible ways to choose a multiset of n elements, in which each symbol can be repeated at most n times.

This is exactly the problem we need to solve to divided all the possible sequences χ^n into classes of sequences having same symbols and same probability.

For example, given the set of symbols $\chi = \{A, B, C, D\}$ these are all the possible multisets of 3 elements we can compute:
AAA
AAB
AAC
AAD
ABB
ABC
...
DDD

It is possible to calculate the number of multisets, for an alphabet of m symbols and a size n, using the formula for the multiset coefficient: $\left(\!\binom{m}{n}\!\right)$.

A well known formula from combinatorics allows us to convert a multiset coefficient into a binomial one:

$$\left(\!\binom{m}{n}\!\right) = \binom{n+m-1}{n}$$

Applying the formula to the AEP, for a given alphabet of symbols of size m, and a fixed sequence size n, we can partition the set of all possible sequences $\{s | s \in \chi^n\}$ in classes of equivalence under the permutation of symbols.

With an abuse of notation, we associate to each class C_i a probability which we define as the probability of an arbitrary sequence:

$$p(C_i) := p(s), s \in C_i$$

To each class we also associate a representative element $Rep(C_i)$, which we define as the first sequence in C_i in lexicographic order:

$$Rep(C_i) = MinLex(\{s \in C_i\})$$

Note that although the classes differ from each other in the symbols composing their sequences, different classes can have the same associated probability. For example, different symbols have the same probability in the chosen distribution, so substituting a symbol with another one having the same probability doesn't change the probability of a class.

Also note that the cardinality of a class depends on the number of repeated symbols in the sequences composing a class. The cardinality of a class varies from 1 if only a single symbol is used, to $n!$ for sequences of length n where no symbol is repeated more that 1 time.

A very simple approach to generate the classes of sequences we need, to fill our typical set, consists in generating all possible multiset of symbols, and then sorting them according to their probabilities.

A wide known algorithm, used to generate all possible multiset of given symbols, in lexicographic order is 2.

Input: An integer m of total symbols, an integer n of multiset size.
Output: Multisets of size n, of m distinct symbols, in lexicographic order.

Procedure:

let a and b be arrays of integers, of size n.
let j and k be integers.

set $a := \{0, 0, ..., 0\}$;
set $b := \{m-1, m-1, ..., m-1\}$;

while *true* **do**
    emit multiset a;

    $j := m - 1$;
    while $a[j] = b[j]$ **do**
        $j := j - 1$;
    end

    if $j < 0$ **then**
        terminate the loop;
    end

    $a[j] := a[j] + 1$;

    $k := j + 1$;
    while $k < n$ **do**
        $a[k] := a[j]$;
        $k := k + 1$;
    end
end

Algorithm 2: Generate all multisets of symbols, in lexicographic order

For the computational complexity of the algorithm we note that in each pass, the algorithm is able to generate the next multiset from a given one, in at most $O(n)$ steps, where n is the size of the multiset.

Applied to our problem, the algorithm is able to generate all possible classes of sequences, the only quirk is that it gives us multisets of index of elements, and we need an additional array to map the index to the actual value of elements.

If we fill an array with the multisets generated by the algorithm, and apply an addition sorting phase using a sorting algorithm based on element comparison (e.g. mergesort, quicksort, heapsort, ...), we can obtain a list of classes of sequences in decrescent order of probability.

Note that for large alphabets or sequence lengths this method is not efficient.

Permutation of multisets

Now that we have the list of classes of sequences, we need to expand them into actual sequences, to fill our typical set.

We can begin to expand the most probable class of sequences, and iterate until we have enough sequences to obtain a typical set with the properties we need.

To "extract" sequences from a class we need to permute the representative element of the class, using a multiset permutation algorithm.

There are many recent advances on the theory of generating permutations of a multiset in a loopless way, and using only limited memory. Some of there results are listed in the bibliography [5] [6].

But although these results are interesting, we opted for an older and simpler algorithm. This was in part because it was simple and more understandable, in part because we need to use the generated partitions in a way that nullifies the asymtotic advantage of more recent methods, when analysing a possible complete program as a whole.

We include here algorithm 7.2.1.2 L 3, described by Knuth in The Art of Computer Programming

vol. 4a [7].

Input: An array of integers a of length n representing a multiset, sorted in crescent order of symbol indexes.
Output: Permutations of the multiset a, in lexicographic order.

Procedure:

let j and k be integers.

```
while true do
    emit permutation a;

    j := n − 2;
    while j >= 0 and a[j] >= a[j + 1] do
        j := j − 1;
    end

    if j < 0 then
        terminate the loop;
    end

    k := n − 1;
    while a[j] >= a[k] do
        k := k − 1;
    end

    swap a[j] and a[k];

    j := j + 1;
    k := n − 1;
    while j < k do
        swap a[j] and a[k];
        j := j + 1;
        k := k − 1;
    end
end
```

Algorithm 3: Generate all multiset permutations, in lexicographic order

This algorithm fits nicely with our choice of representative elements for classes of sequences (i.e. the first sequence in lexicographic order for the class).

The algorithm we described before to generate all possible classes, already generates representative elements which can be used as input for the permutation algorithm.

Regarding the computation complexity of a single pass of the algorithm: we assume a safe upper

bound of $O(n)$ steps to generate the next permutation from a given one, where n is the length of the sequence to permute.

It is possible to prove a stronger bound for the average case, as illustrated by knuth in the solution of exercise 6 of section 7.2.1.2, but the proof is somewhat involved, and we leave it to the literature [7].

Calculating the probabilities for symbols of the alphabet

To sort the class of sequences according to their probabilities, we need to know the probability for each symbol contained in the file we want to compress.

In the case of a fixed length file, we just need to count the occurrences of symbols.

The simplest way to do this, consists in allocating an array p of integers of length $m = |\chi|$, the size of the alphabet of symbols used by the file (e.g. $m = 256$ if symbols are bytes). Then the file to compress is read, and for each symbol $s \in \chi$ encountered, the entry $p[s]$ is incremented by 1.

When the end of file is reached, the array p will contain the probability distribution for the symbols of the alphabet. To reduce memory usage, symbols with a zero probability can be removed from the alphabet.

This approach works well for when the number of symbols is around 2^24 or below. If the alphabet size is larger, a more suitable way is to use an hash table or a linear sorting algorithm such as radix sort.

Final remarks

This appendix illustrated the beginning of a research we are currently still developing.

We mentioned only with some initial problems for a combinatorial approach, and as we have seen the computational complexity is very high. For an exahustive algorithm that generates all possible sequences first, and then sort them, the complexity is not polinomial.

For such an approach to work, it is necessary to generate the possible classes of sequences in decreasing order of probability. This bind the number of classes required to the probability distribution of the alphabet.

If the probability distribution of the symbols is sufficiently unbalanced, we are able to collect a typical set, having the desired probability and epsilon, considering only a fraction of all possible classes of sequences.

Also the space and time complexity of the algorithm should be polynomial in the number of sequences generated by the algorithm up to that point.

We believe such an algorithm can be devised or adapted from techniques already existing, and that it could be applicable in particular real world cases.

Such algorithm will be illustrated in a future document.