

Development and analysis of a new lossless data compression algorithm

Rory Flynn
Newbridge College

Stand No:

Topic	Page No.
Abstract	4
Introduction	5
Glossary	7
Method	8
Results	13
Conclusion and Discussion of Results	14
References	15
Appendix: Results	16

Abstract

My project “Development and analysis of a new lossless data compression algorithm” aimed to create a new lossless data compression algorithm and compare it to current algorithms. Lossless data compression plays an important role in modern digital technology, enabling a media-rich internet. I created a new algorithm, called SIC, that uses Average XORs to bias a file towards 0, and an Arithmetic Coder to exploit this bias and compress the file. I implemented SIC in the C programming language and analysed its performance on a variety of test files. I found that SIC performed greater compression on “artificial” test files than current algorithms, and showed good pattern-finding ability. For most other test files, other current algorithms performed greater compression than SIC. The project achieved its aim of developing and analysing a new lossless data compression algorithm.

Introduction to Data Compression

Data compression is the encoding of data in order to reduce its size. It plays a vital role in modern digital technology, enabling the media-rich Internet we use every day.

Lossy data compression is generally used for multimedia formats where unnecessary information from a file is discarded that should be imperceptible to humans.

For example, videos on YouTube undergo lossy video compression before distribution. Information in the video, that is mostly imperceptible to the viewer unless they analyse the picture closely, is removed – e.g. motion blur, fast-moving content. This is the effect of lossy data compression – unnecessary information that is imperceptible to the viewer is thrown away.

Lossless data compression, the focus of this project, exploits redundancies in the binary representation of a file in order to compress it. Lossless compression does exactly what its name suggests – it compresses files without losing any information. The original file is reproduced exactly from the compressed file.

There has been little evolution of lossless data compression algorithms in recent years - gzip (GNU Zip) came out in 1993, 7-Zip around 1999 and bzip2 in 2000. The gzip project homepage[1] even mentions Y2K compatibility! Gzip, 7-Zip, bzip2 etc. implement compression algorithms even older than themselves. Gzip's LZ77 was published in 1977!

This lack of additional development can be attributed to a combination of:

- Patent restrictions
- “Freezing” of compression formats
- Limits on compression

Patent Restrictions

When many lossless data compressors were being designed, the algorithms they used were still under patent. The .gif image format was tainted by a patent dispute[2] over its use of the LZW (Lempel-Ziv-Welch) compression algorithm. Software developers at the time were unwilling to take risks by using newer algorithms that could be covered by obscure patents.

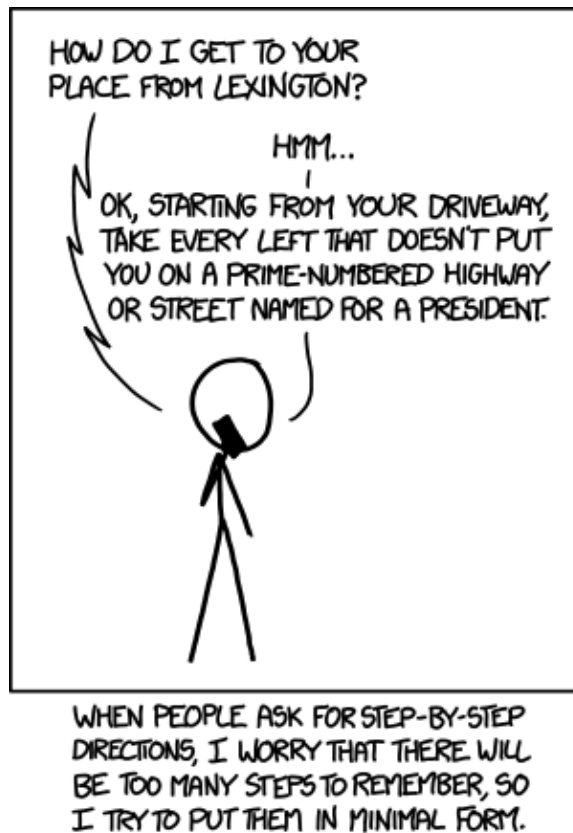
“Freezing” of compression formats

In order to become an accepted standard, a compressor needs to remain fixed in format. Changing the format of gzip drastically for example would cause bad fragmentation, meaning users with previous versions would be unable to decompress a newer format.

Limits of compression

A file's limit of compression is known as its Kolmogorov Complexity. True random data should have a Kolmogorov Complexity of its own size[3] – there should be no way to compress it. This also excludes the possibility of recursive compression – compressed files should have a Kolmogorov Complexity of their own length as well.

"Kolmogorov Complexity", xkcd.com/1155



Random Data and Incompressibility

It is provable through the pigeonhole principle, that it is impossible to universally compress every file by even one bit:

$$2^n > 2^{n-1}$$

n = length of a file

For example, we are given every 5-bit file possible, which is $2^5 = 32$ unique files total.

Let's try to compress all 32 unique 5-bit files down to 4-bit files each.

The set of 4-bit compressed files is $2^4 = 16$ unique files.

$32 > 16$. We cannot get 32 unique 5-bit files from 16 unique 4-bit compressed files. Therefore it is impossible to compress every 5-bit file down to 4 bits or less.

Lossless compression exploits repetition and redundancy in files to compress them, but random data's job is to not have any repetition or redundancy (hence, *random*). Therefore the two are incompatible, and random data is generally incompressible. It is not impossible, but highly improbable, that generated true random data will be compressible

Glossary

XOR operator

I will be referring to the XOR operator throughout, as it is an important tool for this project. Its output is the difference between two series of bits (base 2, binary data). Below, '^' represents the XOR operator. In the XOR of **a** and **b** below, '1' represents that the bit at that position is different, and '0' represents that the bit is the same. For example the first bit in **a** is 1, but the first bit in **b** is 0. They are different, so the XOR bit is 1. The last bit in a and b are both 1, so the XOR bit is 0, meaning the same.

a	11101001
b	01001001
a^b=c	10100000

This transformation is triangular, and so by XORing any two of 'a', 'b', or 'c', we can get the third. We can think of XOR as being reversible.

c	10100000
a	11101001
c^a=b	01001001

Method

Summary of Method

For this project I developed an algorithm called SIC. SIC uses Average XORs to bias files towards 0. SIC then uses an Arithmetic Coder to exploit this bias and compress the file. I have implemented SIC in C.

Average XOR

Before developing SIC, I experimented with basic compression algorithms. One of these was run-length encoding. For example:

```
aaaaaaaaaabbccccccccccccccbaaa
```

could be encoded as:

```
11 ( a ) 14 ( b ) 3 ( a )
```

Similarly, in binary,

```
00000000000000000000001111111111110000011111
```

could be encoded as:

```
21 ( 0 ) 14 ( 1 ) 5 ( 0 ) 5 ( 1 )
```

I noticed that XORing similar series of bits together produced an XOR that was biased towards '0'. (See Glossary explanation of XOR. '0' represents a same bit. Series of bits that are mostly different are biased towards '1'.)

For example, the run-length encoding of

```
0011001100110011001100110101
```


would be

```
2(0)2(1)2(0)2(1)2(0)2(1)2(0)2(1)2(0)2(1)2(0)2(1)1(0)1(1)1(0)1(1)
```

But if we use an average XOR of 0011

Original	0011	0011	0011	0011	0011	0011	0101
XOR	0011	0011	0011	0011	0011	0011	0011
Result	0000	0000	0000	0000	0000	0000	0110

Then we can extend run-length encoding to incorporate average XOR,

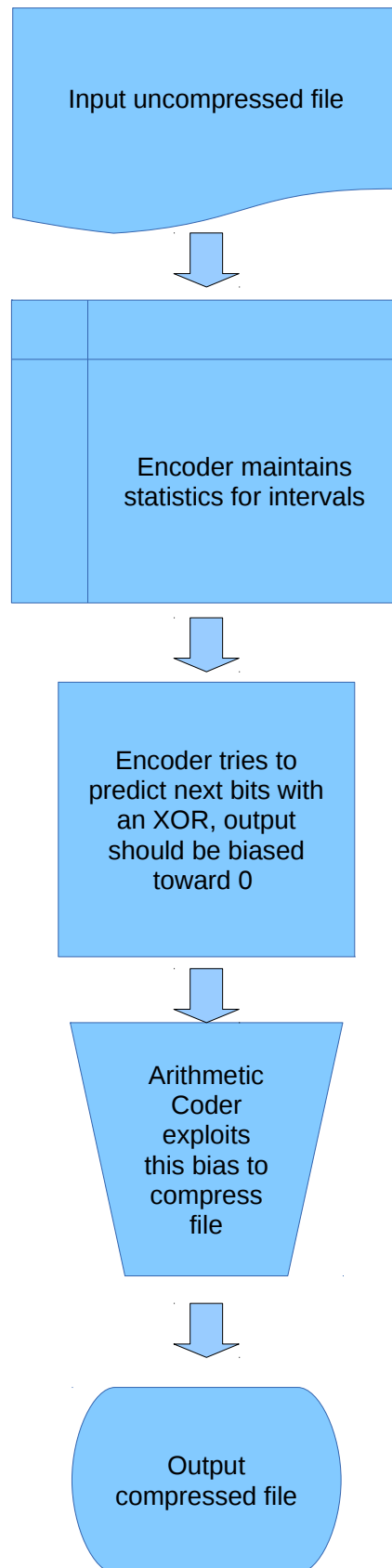
```
X(0011)24(0)1(0)2(1)1(0)
```

Where X indicates Average XOR.

Note how as the Average XOR was close to Original, Result is biased towards 0.

I found that using Average XOR with run-length encoding wasn't efficient enough for files where symbols don't repeat as perfectly as in the examples above. We have shown that we can bias the file towards 0. This bias can be exploited by an arithmetic coder.

Fig. 1.1. SIC encode process



Encoder XOR prediction

Every time SIC reads a bit, it updates an array of statistics for various intervals, counting the number of 1's it has seen.

Interval								
1	0							
2	0	0						
3	0	0	0					
4	0	0	0	0				
5	0	0	0	0	0			
6	0	0	0	0	0	0		
7	0	0	0	0	0	0	0	
8	0	0	0	0	0	0	0	0

Encoder reads **110110110110110110110110** .

Mid interval values are calculated as:

$$\frac{n}{2i}$$

where n = file length, i = interval

Interval									Expected Mid-interval	Total Deviation
1	16								12	4
2	8	8							6	4
3	8	8	0						4	12
4	4	4	4	4					3	4
5	3	5	3	3	3				2.4	5
6	4	4	0	4	4	0			2	12
7	3	3	2	2	2	2	2		1.71	4.03
8	2	2	2	2	2	2	2	2	1.5	4

As we can see above, an interval of 3 or 6 gives the highest deviation (12 bits). However, we should prefer an interval of 3 as its statistics are drawn from a larger sample.

Finding the Average XOR is as simple as recording whether each stat is above or below the Expected Mid-Intervals:

```
8    >    6 = True  = 1
8    >    6 = True  = 1
8    >    0 = False = 0
```

Average XOR = 110

To allow someone else to decode our encoded message, we would have to tell them what its Average XOR was. This can get inefficient for large data sets.

Instead we use adaptive encoding. This means both encoder and decoder start with the same Average XOR, and adapt to a common Average XOR as they build statistics.

Implementation

I have implemented SIC in the C programming language. It consists of two parts – the Average XOR process and the Arithmetic Coding function. I wrote the Average XOR process from scratch, and for the Arithmetic Coding function I modified a sample open source Arithmetic Coder written in C by Michael Dipperstein[4] to use 1-bit symbols.

The source code of my implementation of SIC can be downloaded from
<http://github.com/roryflynn/sic>

Analysis

I tested SIC on various file types and lengths. The complete results of these tests are in the Appendices.

I chose many of these test files based on their use in existing compression benchmarks.

Explanation of Test Data

enwik – XML file of the first n bytes of English Wikipedia. Used in the Hutter Prize[5]. I divided this file up into various intervals.

aaa.txt – The letter 'a' repeated one hundred thousand times, as in the Canterbury Corpus[6].

geo – geographical data from the Canterbury Corpus[6]. Non-ASCII-based.

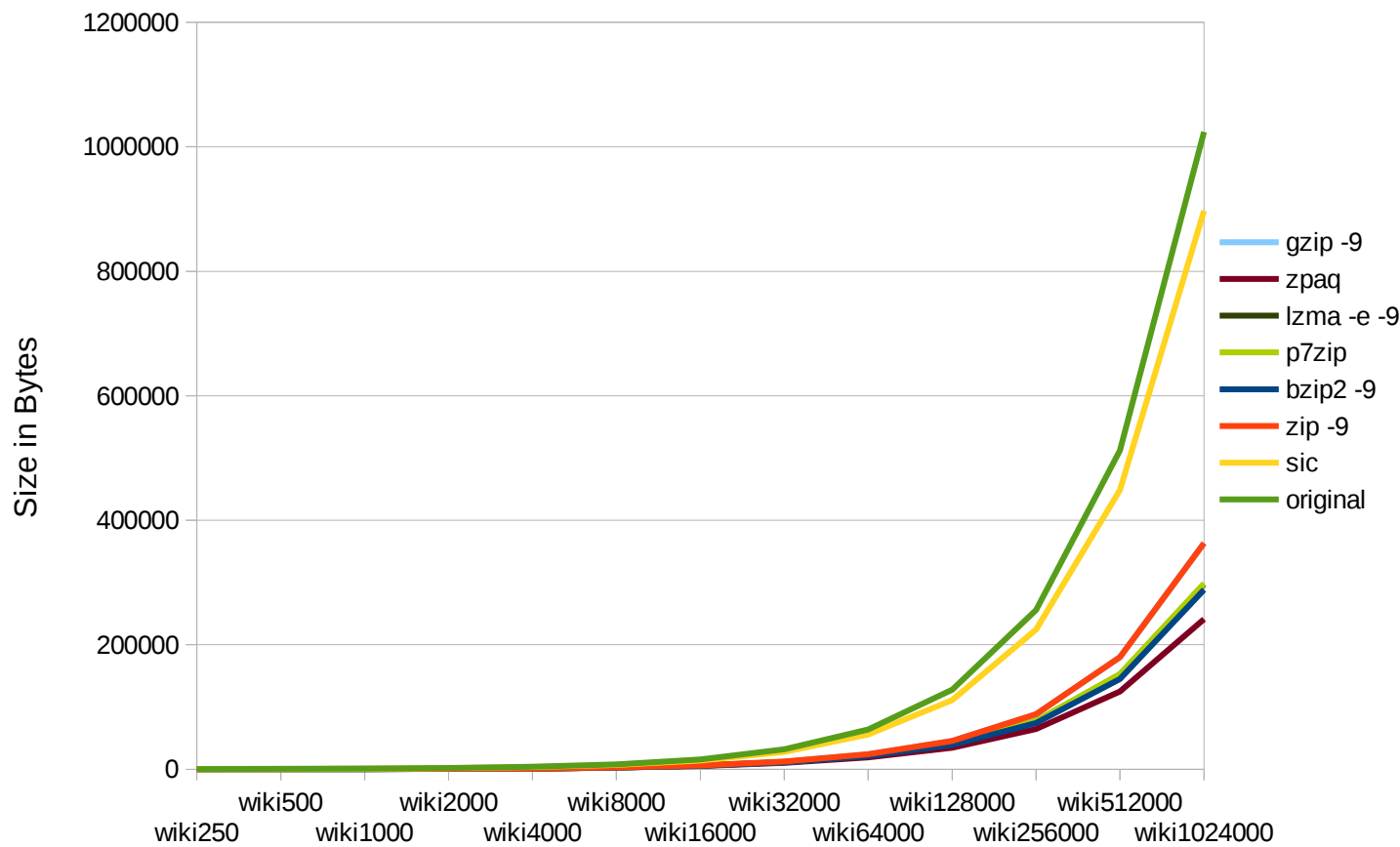
movie – an already-compressed MP4 file, to test SIC's ability to compress low-redundancy data.

great_compressibility – A file I created to test SIC's ability to find patterns in files. Consists of the binary sequence 10011 repeated one thousand times.

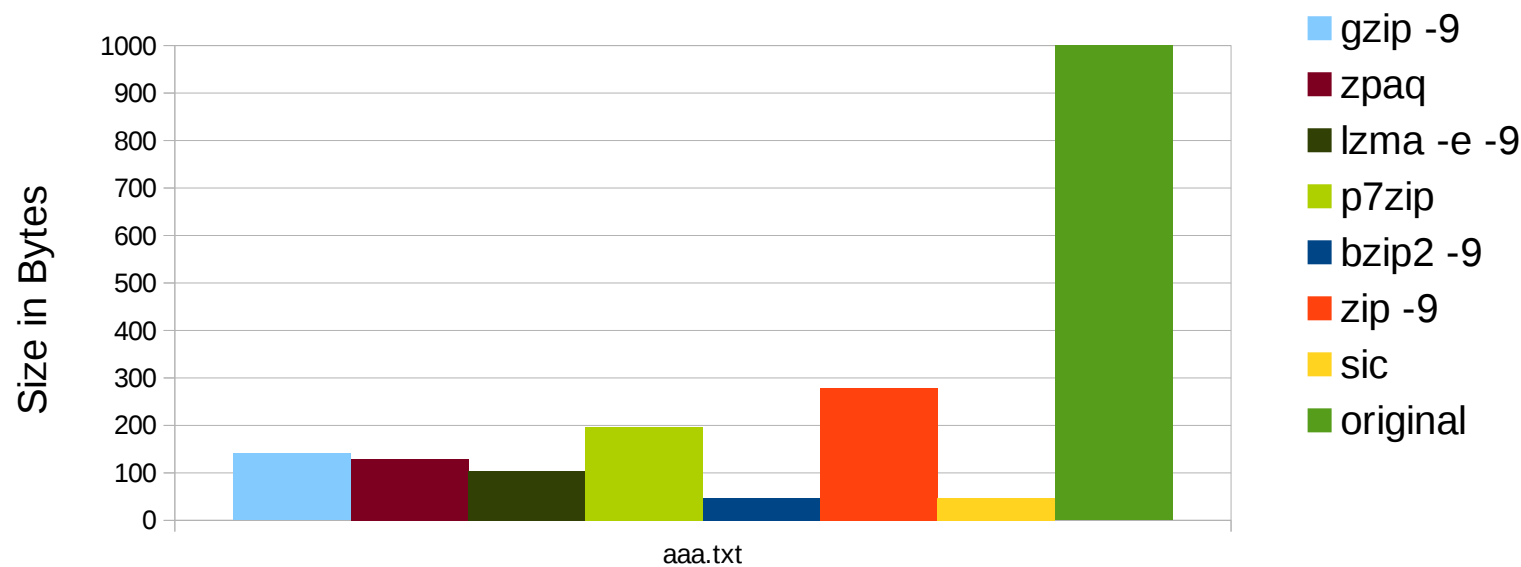
time – Using wiki1024000, to compare the speed of SIC when compressing.

Results

enwik - English Wikipedia in xml format



aaa.txt - 100,000 ASCII a's
(scaled for presentation)



Conclusions and Discussion of Results

As can be seen from the results of **enwik**, SIC performs compression on files, however not to the same extent as current algorithms. This is true for most other tests I ran.

However, for “artificial” data files, like **aaa.txt**, SIC actually performs greater compression than most current algorithms, showing better pattern-finding ability.

SIC also compressed files with patterns of unusual interval lengths such as **great_compressibility** (see appendix), to a greater degree than current algorithms, thanks to this pattern-finding ability.

SIC is currently slower than most current algorithms - see speed benchmark **time** (appendix). SIC would require further optimisation and development to compete with current algorithms.

My project achieved its aim, to develop and analyse a new lossless data compression algorithm.

References

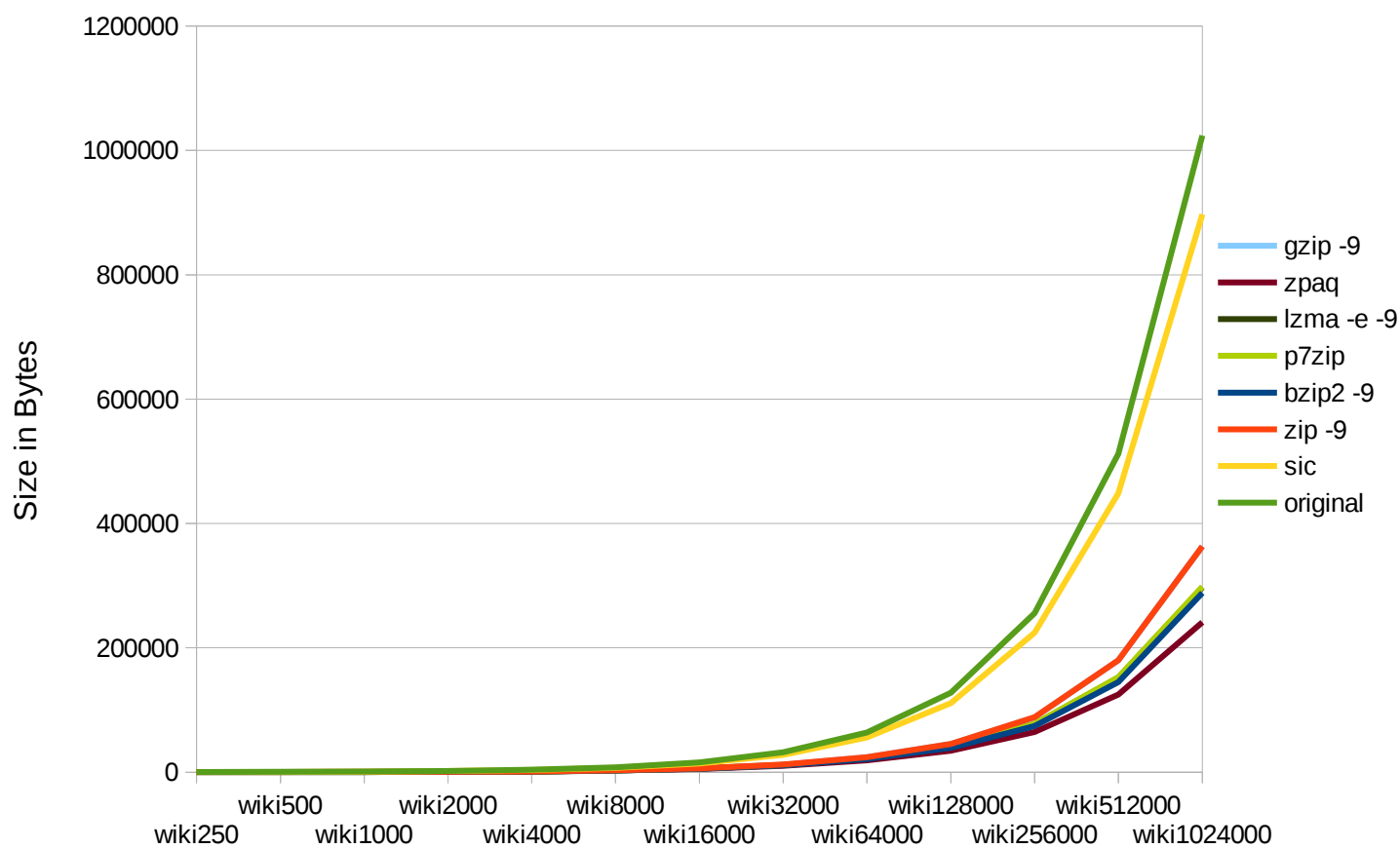
- [1] **gzip.org** – <http://gzip.org>
- [2] **The GIF Controversy: A Software Developer's Perspective** - <http://www.cloanto.com/users/mcb/19950127giflzw.html>
- [3] **Kolmogorov Complexity - A Primer**
<http://jeremykun.com/2012/04/21/kolmogorov-complexity-a-primer/>
- [4] **Arithmetic Coder: Michael Dipperstein** - <http://michael.dipperstein.com/arithmetic/>
- [5] **Hutter Prize** - <http://www.hutter1.net/prize/index.htm>
- [6] **Canterbury Corpus** - <http://corpus.canterbury.ac.nz/descriptions/>

Appendix: Results

enwik – English Wikipedia (compressed size in bytes)

enwik8	wiki250	wiki500	wiki1000	wiki2000	wiki4000	wiki8000	wiki16000	wiki32000	wiki64000	wiki128000	wiki256000	wiki512000	wiki1024000
gzip -9	161	272	345	635	977	2855	6122	12154	24082	45478	88178	180073	363270
zpaq	244	356	426	681	1010	2667	5350	10187	19178	34581	64591	124874	241032
lzma -e -9	156	281	356	630	945	2779	5854	11576	22393	41018	77196	151876	297272
p7zip	252	377	452	735	1060	2894	5967	11692	22514	41198	77675	152683	298531
bzip2 -9	191	310	391	683	1068	2906	5877	11176	21394	39237	74241	144793	288349
zip -9	300	411	484	775	1117	2995	6263	12295	24223	45620	88320	180215	363412
sic	241	467	899	1801	3607	7164	14116	27981	55795	111264	224311	448547	897084
original	250	500	1000	2000	4000	8000	16000	32000	64000	128000	256000	512000	1024000

enwik - English Wikipedia in xml format

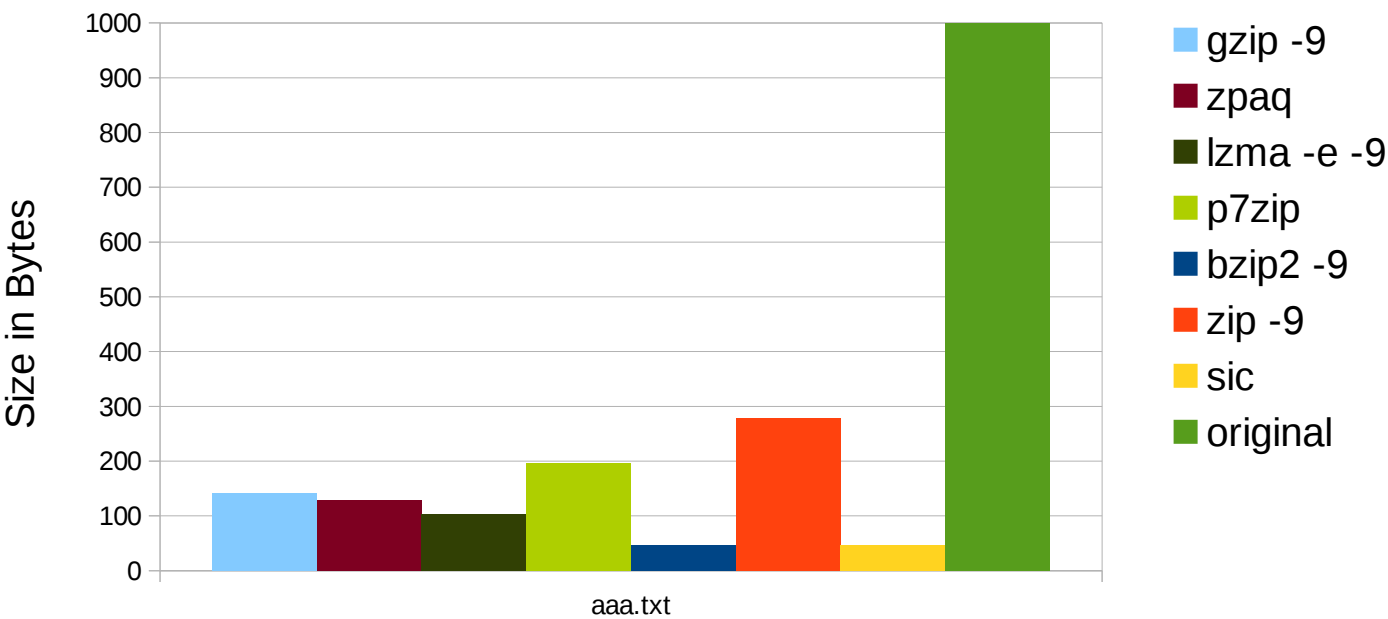


aaa.txt – 100,000 ASCII a's (compressed size in bytes)

	aaa.txt
gzip -9	141
zpaq	129
lzma -e -9	103
p7zip	196
bzip2 -9	47
zip -9	279
sic	47
original	100000

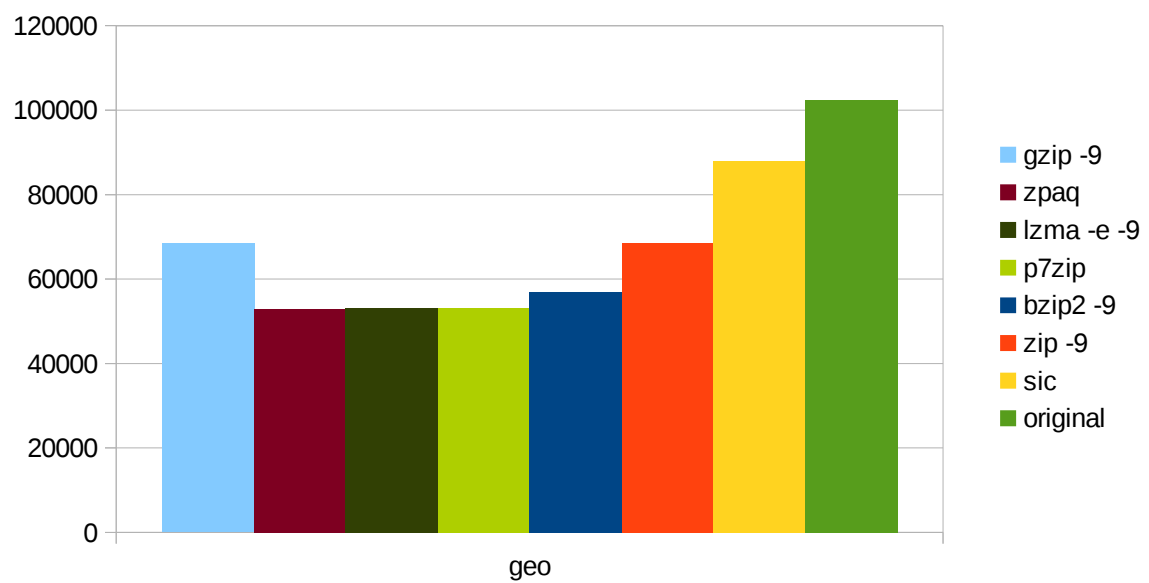
aaa.txt - 100,000 ASCII a's

("original" scaled for presentation)



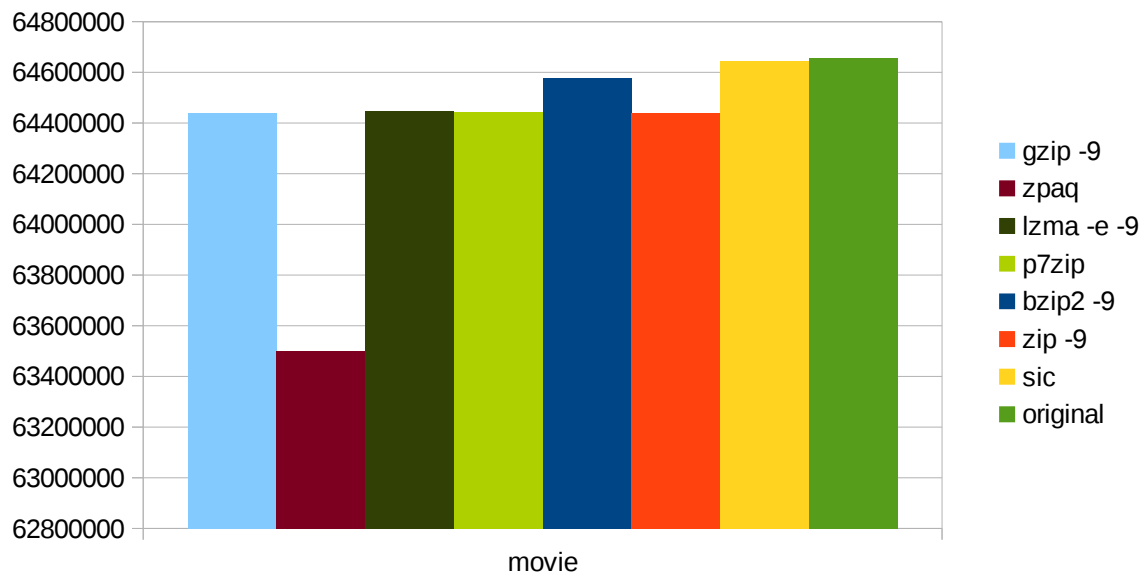
geo – geological data (compressed size in bytes)

	geo
gzip -9	68414
zpaq	52901
lzma -e -9	53121
p7zip	53167
bzip2 -9	56921
zip -9	68548
sic	87985
original	102400



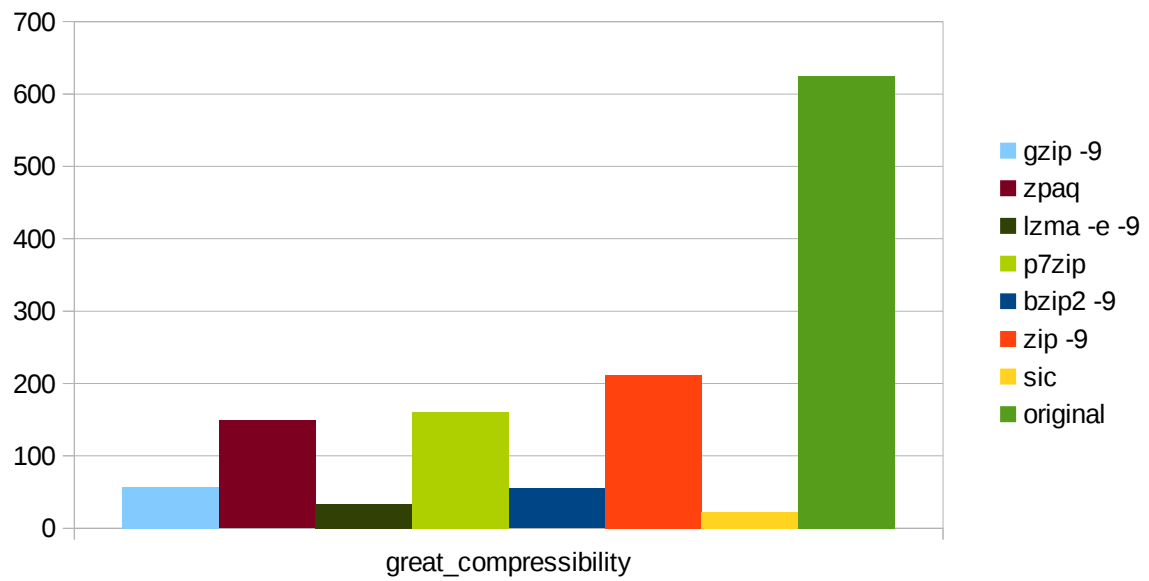
movie – already compressed MP4 (compressed size in bytes)

	movie
gzip -9	64438782
zpaq	63499190
lzma -e -9	64447898
p7zip	64445638
bzip2 -9	64576181
zip -9	64438937
sic	64644522
original	64657027



great-compressibility (compressed size in bytes)

great_compressibility	
gzip -9	57
zpaq	149
lzma -e -9	34
p7zip	161
bzip2 -9	55
zip -9	212
sic	22
original	625



time – compressing wiki1024000 (time taken to compress in seconds)

time	Gzip -9	zpaq	lzma -e -9	p7zip	Bzip2 -9	Zip -9	sic
wiki1024000	0.075	1.579	0.436	0.242	0.108	0.077	1.702

