Technical Report

Dmitry Zbarski

May 2014

Abstract

This is a technical report of tests that were done to compare simple LZ77 algorithm against LZ77 with EAC extension.

Contents

Chapter 1

Introduction

This is a technical report of work that was done to check whether EAC can improve simple LZ77 experimentally. To check this a program was written that implements both simple LZ77 algorithm and LZ77 with EAC extension. This document contains description of all algorithms that were used and programmed, description of program and results of running simple LZ77 and LZ77 with EAC extension on sample files.

List of algorithms that were used:

CFC Comma Free Coding [2]

LZ77 Lempel-Ziv data compression algorithm that maintains sliding window during compression [1]

EAC Entropy Adaptive Coding is an extension for data compression algorithm that maintains sliding window during compression [3]

Chapter 2

CFC - Comma Free Coding

2.1 Overview

Comma free coding [2] is an algorithm that allows coding arbitrary length sequences of bits[1]. This algorithm will be used later by simple LZ77 algorithm to effectively code any data. In simple words the algorithm works as follows. Let k be any positive integer number and let $b(k)$ be a binary expansion of integer k. Further let's define $e(i) = 0^{i-1} * 1$[2]. Then CFC encodes any positive integer number like this $cfc(k) = e(|b(k)|) * |b(k)| * b(k)$. Clearly CFC is bijection between $[1, \infty)$ and $\{0, 1\}*$[3].

cfc_encode This function encodes any given bit string (bits are actually representing some number) using CFC algorithm and returns bit string (a sequence of bits) that represent encoded value. See algorithm 1

cfc_decode This function decodes CFC encoded value stored in a bit string. Return value is actually a pair of decoded value and length of CFC bit string that was detected during decoding process. See algorithm 2

e Supplementary function that gets bit string as parameter and returns bit string consisting of $len(bitString)$ zeroes followed by 1 one. For example for bit string 001011 value returned is 000001. See algorithm 3

e_inverse Supplementary function that gets bit string and returns number of zeroes till first one is met plus one. For example if bit string is 00001, then value returned is 5. See algorithm 4

2.2 Algorithm

Functions that appear in algorithms below, but not described fully as separate algorithms:

- $len(bitString)$ - this function receives as an argument bit string and returns it's length.
- $bitConcat(bitString1, bitString2)$ - this function receives 2 or more bit strings as arguments and returns their concatenation.
- $bitSubStr(bitString, offset, length)$ - this function receives 3 arguments: bitString, start offset and length and returns part of bitString 'length' length that starts at offset.

[1]For full explanation of how this algorithm works look at [2, Appendix Comma-Free Coding of an Unbounded Integer]
[2]$*$ is a concatenation
[3]$*$ is the Kleene star

Algorithm 1 Comma Free Coding - encode

```
1: function CFC_ENCODE($bitString$)
2:     $l \leftarrow len(bitString)$
3:     return $bitConcat(E(l), l, bitString)$
4: end function
```

Algorithm 2 Comma Free Coding - decode

```
1: procedure CFC_DECODE($bitString$)
2:     $l\_len \leftarrow E\_INVERSE(bitString)$
3:     $l \leftarrow bitSubStr(bitString, l\_len, l\_len)$
4:     $output \leftarrow bitSubStr(bitString, l\_len + l\_len, l)$
5:     return $(output, l\_len + l\_len + l)$
6: end procedure
```

Algorithm 3 Comma Free Coding - function e

```
1: function E($bitString$)
2:     $i \leftarrow 0$
3:     while $i < len(bitString) - 1$ do
4:         $output[i] \leftarrow 0$
5:     end while
6:     $output[i] \leftarrow 1$ return $output$
7: end function
```

Algorithm 4 Comma Free Coding - function e^{-1}

```
1: function E_INVERSE($bitString$)
2:     $i \leftarrow 0$
3:     while $bitString[i] \neq 1$ do
4:         $i \leftarrow i + 1$
5:     end while
6:     $i \leftarrow i + 1$ return $i$
7: end function
```

Chapter 3

LZ77 algorithm

3.1 Overview

LZ77 is an algorithm was published in papers by Abraham Lempel and Jacob Ziv in 1977. In tests that were performed algorithm was not used in the exact manner as it was published for the first time [1]. Instead it was implemented as it was described in "The Sliding-Window Lempel-Ziv algorithm is asymptotically optimal." [2]. The latter implementation uses CFC.

In this implementation in compressed data each bit string is coded with it's length encoded using CFC followed by either data itself or an index in sliding window where coded data is found. To determine whether data itself is following length or an index, size of bit string used. If size of bit string less than or equals to $\log(\textit{window size})$, then uncompressed data follows. Otherwise an index in sliding window follows (index is coded using $\log(\textit{window size}$ bits).

To determine window size, first CFC encoded value can be used. As noted above, first 'window size' bits are not compressed. Therefore first CFC coded number indicates what window size was used to compress data. Knowing window size allows us to compute $\log(\textit{window size})$ and distinguish between compressed chunks and uncompressed chunks of data.

lz77_encode given window size and optional training sequence[1], compresses data and returns it as bit string (see algorithm 5).

lz77_decode given data, optional training sequence, optional winSize and optional original size (size of data after decompression) decompresses data and returns pair of decompressed data and length of compressed data that was processed (see algorithm 6).

3.2 Algorithm

This section contains pseudo-code of LZ77 compression and decompression algorithms.

These algorithms were adapted to be able to get training sequence (which is a sliding window that precedes current data). This is to allow later reuse these algorithms in EAC extension4.

- $len(bitString)$ - this function receives as an argument bit string and returns it's length.
- $bitConcat(bitString1, bitString2)$ - this function receives 2 or more bit strings as arguments and returns their concatenation.
- $bitSubStr(bitString, offset, length)$ - this function receives 3 arguments: bitString, start offset and length and returns part of bitString 'length' length that starts at offset.
- $findBestMatch(bitString1, bitString2)$ - returns pair, $offset$ and $length$, of longest bit string in bitString1 that matches $length$ first bits in bitString2.
- $bitAppendNumber(bitString, number)$ - appends $number$'s bit representation to $bitString$.

[1]training sequence - a number of bits that are not compressed in the beginning of data that are needed to create first window

- $bitGetNumber(bitString, offset)$ - return number representation stored at offset $offset$ in bit string $bitString$.
- $NUMBER\_BITS$ - represents how many bits long our numbers.

Following is a pseudo-code of compression (algorithm 5) and decompression (algorithm 6) algorithms.

Algorithm 5 Simple LZ77 compression algorithm

```
function LZ77_ENCODE($winSize, trainingSequence, data$)
    $output \leftarrow new\ bitString$                                  ▷ output is a bit string
    $i \leftarrow 0$                                                    ▷ index in data that we are currently on
    $lastWritten \leftarrow 0$                                          ▷ index in data till where we already wrote
    if $trainingSequence\ is\ empty$ then                               ▷ Write first winSize bits uncompressed
        $output \leftarrow bitConcat(output, cfc\_encode(winSize))$
        for $i \leftarrow 0, winSize$ do
            $output \leftarrow bitConcat(output, data[i])$
        end for
        $lastWritten \leftarrow winSize - 1$
    end if
    while $i < len(data)$ do
        $window \leftarrow bitSubStr(data, i - winSize, winSize)$
        $remainder \leftarrow bitSubStr(data, i, len(data))$
        $(offset, length) \leftarrow findBestMatch(window, remainder)$
        if $length > \log(winSize)$ then                                 ▷ If found match that is long enough, replace it
            if $lastWritten \neq i - 1$ then                            ▷ Write uncompressed bits if needed
                $output \leftarrow bitConcat(output, cfc\_encode((i - 1) - lastWritten))$
                $output \leftarrow bitConcat(output, bitSubStr(data, lastWritten + 1, (i - 1) - lastWritten))$
                $lastWritten \leftarrow i - 1$
            end if
            $output \leftarrow bitConcat(output, cfc\_encode(length))$
            $output \leftarrow bitAppendNumber(output, offset)$
            $i \leftarrow i + length$
            $lastWritten \leftarrow i - 1$
        else                                                            ▷ Match found is too short, continue
            $i \leftarrow i + 1$
        end if
    end while
    return $output$
end function
```

Algorithm 6 Simple LZ77 algorithm decompression

```
 1: function LZ77_DECODE($data, trainingSequence, winSize, outputSize$)
 2:     if $winSize = 0$ then
 3:         $winSize = cfc\_decode(data)$
 4:     end if
 5:     $i \leftarrow 0$                                      ▷ index in data that we are currently on
 6:     $output \leftarrow new\ bitString$                    ▷ output is a bit string
 7:     if $trainingSequence\ is\ not\ null$ then             ▷ prepend training sequence to data
 8:         $data \leftarrow bitConcat(trainingSequence, data)$
 9:         $i \leftarrow len(trainingSequence)$
10:     end if
11:     while $i < len(data)$ do
12:         if $outputSize \neq 0 \wedge len(output) \geq outputSize$ then
13:             break
14:         end if
15:         $(length, cfcLength) \leftarrow cfc\_decode(bitSubStr(data, i, len(data) - i))$
16:         $i \leftarrow i + cfcLength$
17:         if $length > \log(winSize)$ then                  ▷ offset follows
18:             $offset \leftarrow bitGetNumber(data, i)$
19:             $output \leftarrow bitSubStr(output, len(output) - offset, length)$
20:             $i \leftarrow i + NUMBER\_BITS$
21:         else                                              ▷ uncompressed data follows
22:             $output \leftarrow bitConcat(output, bitSubStr(data, i - offset, length))$
23:             $i \leftarrow i + length$
24:         end if
25:     end while
26:     return $(output, i)$
27: end function
```

Chapter 4

EAC extension

4.1 Overview

EAC - Entropy Adaptive Coding, is an extension that should improve any existing sliding window algorithm. EAC splits data into blocks and tries to compress each block with all available window sizes. Window sizes that are tested are powers of 2 from 8 to size of block. For each block, window size that produces best compression ratio is selected. To simplify coding of window size changes, we will simply code $\log(\textit{window size})$ between blocks. Window size of first block can be known from CFC value coded in start of it. [1]

It's worth to note that because data is split into blocks and each block does not depend on output of any other block (except window size change between blocks) it is very easy to parallelize this algorithm. Test program that was built to actually test and that is described in next chapter actually utilizes this fact to improve performance.

First block of data is used as training sequence and therefore is not compressed. Later each block uses data from previous block for training sequence. Therefore sliding window, when starting compression of each block, lies in previous block. As compression continues, sliding window slides gradually from previous block into current.

After compression of block is complete it can be written to file only after all previous blocks were compressed. This is because prior to finishing compression of previous blocks, we do not know their size. In this chapter we will only look at sequential version of algorithm to simplify things and ease understanding of algorithm. But in test program, algorithm was actually parallelized to improve performance and speed up tests of files.

eac_encode given data and block size compresses data and returns it as bit string (see algorithm 7

eac_decode given data decompresses it and returns as bit string 8

4.2 Algorithm

This section contains pseudo-code of EAC compression and decompression algorithms.

List of functions used in these algorithms, which implementation is out of scope of this report:

- $len(bitString)$ - this function receives as an argument bit string and returns it's length.
- $bitConcat(bitString1, bitString2)$ - this function receives 2 or more bit strings as arguments and returns their concatenation.
- $bitSubStr(bitString, offset, length)$ - this function receives 3 arguments: bitString, start offset and length and returns part of bitString 'length' length that starts at offset.
- $bitAppendNumber(bitString, number)$ - appends $number$'s bit representation to $bitString$.

[1] For details why, see chapter LZ77 algorithm

- $bitGetNumber(bitString, offset)$ - return number representation stored at offset $offset$ in bit string $bitString$.
- $split(bitString, size)$ - splits $bitString$ bit string into multiple bit strings where each bit string's length is $length$ bits long (except may be last bit string).
- $shift(array)$ - returns pair of first element of array and array without first element.
- $NUMBER\_BITS$ - represents how many bits long our numbers.

Following is a pseudo-code of compression (algorithm 5) and decompression (algorithm 6) algorithms.

Algorithm 7 EAC compression algorithm that uses LZ77 algorithm

```
procedure EAC_ENCODE($blockSize, data$)
    $blocks \leftarrow split(data, blockSize)$
    $output \leftarrow new\ bitString$
    $(first, data) \leftarrow shift(blocks)$
    $i \leftarrow 8$
    $best \leftarrow new\ bitString$
    while $i \leq blockSize$ do
        $tmp \leftarrow lz77\_encode(i, NULL, first)$
        if $len(tmp) < len(best)$ then
            $tmp \leftarrow best$
        end if
        $i \leftarrow i \times 2$
    end while
    $output \leftarrow bitConcat(output, best)$
    $prev \leftarrow first$
    for $block \in blocks$ do
        $best \leftarrow new\ bitString$
        $i \leftarrow 8$
        $winSize \leftarrow 8$
        while $i \leq blockSize$ do
            $tmp \leftarrow lz77\_encode(i, prev, block)$
            if $len(tmp) < len(best)$ then
                $best \leftarrow tmp$
                $winSize \leftarrow i$
            end if
            $i \leftarrow i \times 2$
        end while
        $output \leftarrow bitAppendNumber(output, winSize)$
        $output \leftarrow bitConcat(output, best)$
        $prev \leftarrow block$
    end for
    return $output$
end procedure
```

Algorithm 8 EAC decompression algorithm that uses LZ77 algorithm

data,trainingSequence,winSize,outputSize

```
1:  procedure EAC_DECODE($blockSize, data$)
2:      $output \leftarrow new\ bitString$
3:      $trainingSequence \leftarrow new\ bitString$
4:      $winSize \leftarrow 0$
5:      while $len(data) \geq 0$ do
6:          $(tmp, length) \leftarrow lz77\_decode(data, trainingSequence, winSize, blockSize)$
7:          $output \leftarrow bitConcat(output, tmp)$
8:          $trainingSequence \leftarrow tmp$
9:          $data \leftarrow bitSubStr(data, length, len(data) - length)$
10:         $winSize \leftarrow bitGetNumber(data, 0)$
11:         $data \leftarrow bitSubStr(data, NUMBER\_BITS, len(data) - NUMBER\_BITS)$
12:     end while
13:     return $output$
14: end procedure
```

Chapter 5

Test program

5.1 Availability

Sources of program, scripts and test files available at https://github.com/dintel/eac.

5.2 Overview

For testing purposes a program written in C was developed that implements both LZ77, as described in chapter LZ77 algorithm and EAC as described in chapter EAC extension. C programming language was chosen because of performance reasons and because C programming language allows relatively easy working with bit strings.

Program was used on different files to test how both LZ77 and EAC perform in terms of compression ratio. To ensure that every run of program was correct, files were decompressed in the end to ensure that original file can be restored. Test of files were run by scripts written in bash. In the end results were stored in simple CSV[1] file.

Data from this file, result.csv, was then used to generate all graphs using gnuplot[2] program in Partial list of graphs. Generation of image files is done by script that is written in PHP programing language. Additionally a simple interactive web interface was developed to ease inspection of graphs without generating final graph images.

5.3 Modules

5.3.1 bit_string

This module implements a bit string. Bit string is a data type that consists of multiple ordered bits.

List of functions that this module provides:

bit_string_init - Initialize new bit string

bit_string_destroy - Destroy bit string

bit_string_cmp - Compare 2 bit strings

bit_string_sub_cmp - Compare 2 bit substrings

bit_string_count_zeroes - Count number of 0 bits

bit_string_append_bit - Append bit to bit string

bit_string_get_bit - Get value of bit at some index

[1]CSV - Comma Separated Values. Simple text format to store table data. In this case each value in row was separated by semicolon (;).

[2]More information about gnuplot see at their official website http://www.gnuplot.info

bit_string_substr - Get substring of bit string

bit_string_concat - Append one bit string to another

bit_string_concat_and_destroy - Append one bit string to another and destroy source bit string

bit_string_copy - Copy substring of bit string to another bit string

bit_string_read_byte - Read up to 8 bits into byte

bit_string_print - Print bit string

bit_string_full_copy - Copy bit string into another bit string

For detailed information about this module, see bit_string module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.2 bit_string_writer

This module implements bit string writer that allows writing bit string into file.

- **bit_string_writer_init** - Initialize new bit string writer
- **bit_string_writer_destroy** - Destroy bit string writer
- **bit_string_writer_write** - Write bit string
- **bit_string_writer_flush** - Flush bit string writer buffer onto disk
- **bit_string_writer_write_byte** - Write some bits of byte using bit string writer

For detailed information about this module, see bit_string_writer module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.3 block

This module is used only by EAC compressor/decompressor. It provides abstraction of block of data that EAC works with.

List of functions that this module provides:

- **block_init** - Initialize new block
- **block_destroy** - Destroy block
- **block_update** - Update block result
- **block_is_complete** - Check whether block is completed

For detailed information about this module, see block module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.4 cfc

This module provides CFC related functions.

List of functions that this module provides:

- **cfc_encode** - Encode number using CFC
- **cfc_decode** - Decode CFC encoded number

For detailed information about this module, see cfc module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.5 delta

This module provides abstraction of how window size changes are read/written between blocks.
List of functions that this module provides:

- **nw_change_encode** - Output encoded change in window size
- **nw_change_decode** - Decode window size of next block

For detailed information about this module, see delta module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.6 log

This module provides macros related to logging when some variable is set to 1.
List of macros that this module provides:

- **PRINT_VERBOSE** - Macro that outputs message only if log_verbose flag is set
- **PRINT_DEBUG** - Macro that outputs message only if log_debug flag is set

For detailed information about this module, see log module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.7 lz77

This module provides functions that implement LZ77 algorithm as it is described in chapter LZ77 algorithm
List of functions that this module provides:

- **lz77_encode** - Encode block bit string using LZ77
- **lz77_decode** - Decode block bit string using LZ77

For detailed information about this module, see lz77 module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.3.8 queue

This module provides functions related to queue of compression jobs. Each job tries to compress block 5.3.3 of data using some window size. Upon completion block is updated with results of compression. Jobs run asynchronously and number of max threads can be specified when running queue.
List of functions that this module provides:

- **queue_init** - Initialize new queue of jobs
- **queue_destroy** - Destroy queue of jobs
- **queue_run_job** - Run new job
- **queue_add_job** - Add new job to queue
- **queue_fetch_finished** - Fetch first finished job
- **queue_run** - Run queue
- **queue_destroy_job** - Destroy job

For detailed information about this module, see queue module documentation in project documentation (for details how to access project documentation see chapter Project documentation).

5.4 Executable programs

5.4.1 eac_encode

```
Usage: eac_encode [OPTION...]  -i INPUT_FILE -o OUTPUT_FILE
Entropy Adaptive Coding - encoder

  -b, --block-size=BLOCK_SIZE   Block size
  -d, --debug                  Don't produce any output
  -e, --eac                    Use Adaptive Entropy Coding
  -i, --input=FILE             Input file
  -n, --window-size=FILE       LZ77 window size (ignored when --eac is used)
  -o, --output=FILE            Output file
  -t, --threads=THREADS        Number of concurrent threads
  -v, --verbose                Produce verbose output
  -?, --help                   Give this help list
      --usage                  Give a short usage message
  -V, --version                Print program version
```

eac_encode program can compress files using either LZ77 or EAC algorithms. By default eac_encode compresses using LZ77 algorithm. If EAC is desired, then –eac switch must be specified.

Additional output verbosity can be enabled by –verbose option. If that's not enough, specifying –debug option will enable debug output.

When compressing using EAC algorithm, multiple threads can be used. –threads parameter controls how many threads should be used during compression process.

By default eac_encode produces one line of output where following values are separated by semicolon (;):

1. **File size** - size of input file in bits
2. **Compressed size** - size of output file in bits
3. **Ratio** - compression ratio
4. **File longest match** - length of longest match found in file during compression
5. **Average longest match** - average between longest matches in each block (in case LZ77 used equals to previous value)
6. **Standard deviation of longest match** - Standard deviation of longest matches of all blocks (in case LZ77 used equals to 0)
7. **Block longest match** - space separated list of lengths of longest matches in each block

5.4.2 eac_decode

```
Usage: eac_decode [OPTION...]  -i INPUT_FILE -o OUTPUT_FILE
Entropy Adaptive Coding - decoder

  -b, --block-size=BLOCK_SIZE   Block size
  -d, --debug                  Don't produce any output
  -e, --eac                    Use Adaptive Entropy Coding
  -i, --input=FILE             Input file
  -o, --output=FILE            Output file
  -v, --verbose                Produce verbose output
  -?, --help                   Give this help list
      --usage                  Give a short usage message
  -V, --version                Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

eac_decode program can decompress files using either LZ77 or EAC algorithms. By default eac_decode decompresses using LZ77 algorithm. If EAC is desired, then –eac switch must be specified.

Additional output verbosity can be enabled by –verbose option. If that's not enough, specifying –debug option will enable debug output.

By default eac_decode produces one line of output where following values are separated by semicolon (;):

1. **Compressed size** - size of input file in bits
2. **Original size** - size of output file in bits
3. **Ratio** - compression ratio

5.4.3 generator

```
Usage: generator [OPTION...]  -o OUTPUT_FILE -n SIZE
Entropy Adaptive Coding - generator

  -n, --size=FILE            Size of file in bytes
  -o, --output=FILE          Output file name
  -p, --probability=NUM      Probability of 0 (probability of 1 is calculated
                             automatically)
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

This is a supplementary program that was used to create some test files. It generates files of given size where probability of 0 and 1 bits is as specified by -p parameter.

5.5 Scripts

5.5.1 tests/graphgen.php

```
Usage: ./graphgen.php <result.csv>
```

This script is written in PHP programming language and requires PHP interpreter to be installed. Additionally it requires gnuplot to be installed. It parses result.csv file and generates graphs of results in tests/images directory.

5.5.2 tests/jsonReport.php

```
Usage: ./jsonReport.php
```

This script generates file viewer/result.json that is later used by Web based interactive graph viewer.

5.5.3 tests/performance.sh

```
Usage: ./performance.sh
```

This script is written in bash. For each file located in tests/files directory it runs perf_single.sh script to test compression and decompression using LZ77 and EAC with windows sizes from 8 to 32768 and block sizes 64 to 65536 [1]. Tests are run in parallel, so that all tests for any single file are started simultaneously and then script waits till they all finish. After all tests are complete all result files in results directory combined into result.csv file [2].

5.5.4 tests/perf_single.sh

```
Usage: ./perf_single.sh <file> <block-size> <window-size> [-e]
```

[1] window size and block size are doubled at each iteration

[2] previous result.csv is renamed into result-DATE.csv where DATE is date when result.csv was created

This script compresses and decompresses specified file using:

1. If -e specified then EAC algorithm with specified block size.
2. If -e not specified then LZ77 algorithm with specified window size.

After completing it's run following values separated by semicolon (;) are output into tests/results/FILE-BLOCK_SIZE-WINDOW_SIZE.ALGORITHM.result (where FILE is file name, BLOCK_SIZE is block size, WINDOW_SIZE is window size and ALGORITHM is either eac or lz77)

1. **File name** - name of file
2. **Block size** - block size
3. **Window size** - window size
4. **Algorithm** - algorithm that was used during compression/decompression
5. **Encode time** - time took to compress file
6. **Decode time** - time took to decompress file
7. **Result** - SUCCESS in case input and decompressed files match
8. **File size** - size of input file in bits
9. **Compressed size** - size of output file in bits
10. **Ratio** - compression ratio
11. **File longest match** - length of longest match found in file during compression
12. **Average longest match** - average between longest matches in each block (in case LZ77 used equals to previous value)
13. **Standard deviation of longest match** - Standard deviation of longest matches of all blocks (in case LZ77 used equals to 0)
14. **Block longest match** - space separated list of lengths of longest matches in each block

5.5.5 tests/sanity.sh

```
Usage: ./sanity.sh
```

This is a simple sanity test script. It simply compresses and then decompresses each file in tests/files directory such that it's size is between 32KB and 50KB. Compression and decompression is done using both LZ77 and EAC algorithms. After decompression decompressed file is compared to original file. If files match, then script prints SUCCESS - FILENAME.ALGORITHM (where FILENAME is name of file tested and ALGORITHM is algorithm used - eac or lz77). Otherwise script prints FAILED - FILENAME.ALGORITHM.

This script is used to run simple sanity tests of eac_encode and eac_decode. Each output line of this script must begin with SUCCESS. If output contains line beginning with FAILED, then there is a bug in a program[1].

5.6 Additional parts

5.6.1 Interactive Web based graphs viewer

Interactive web based graphs viewer is located in viewer directory. To work correctly following requirements must be met:

- tests/performance.sh script must be run to generate tests/result.csv file.

[1]Failure can be caused by limitation of eac_encode and/or eac_decode programs

- tests/jsonReport.sh script must be run to generate viewer/result.json file.

To launch web based graphs viewer just open viewer/index.html file in your browser. You will then be able to see graphs in your browser. On top of page there is a selector of test file whose results are shown.

5.6.2 Build system

Requirements for building executable files:

- GNU Make (version $\geq$ 4.0)
- GNU C Compiler (version $\geq$ 4.8.2)

Project uses GNU Make to build all executable files. All source files compiled using GNU C Compiler. To build all executable files run *'make all'* command in project directory. This will build all executable files: eac_encode, eac_decode, generator.

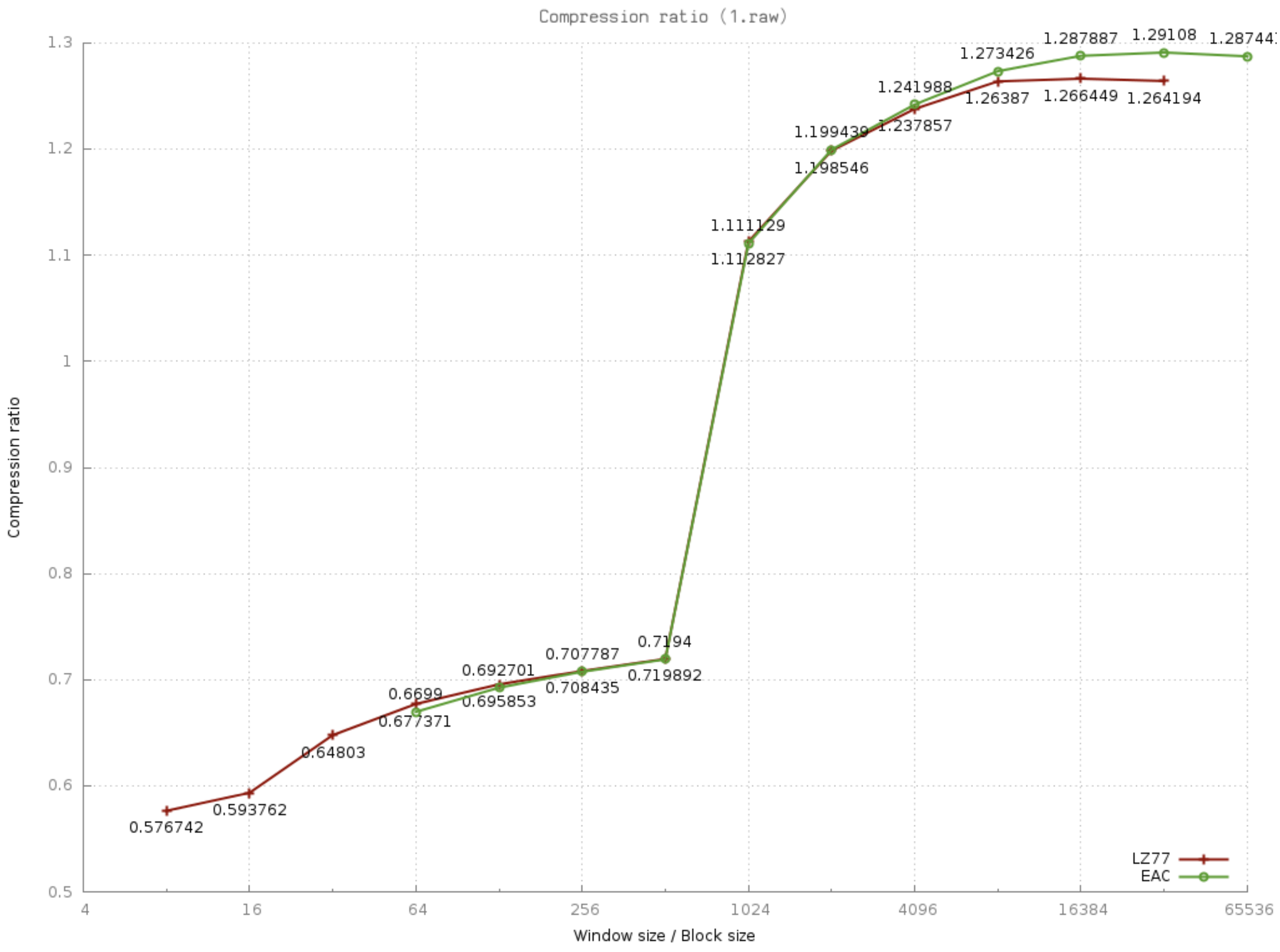
5.6.3 Project documentation

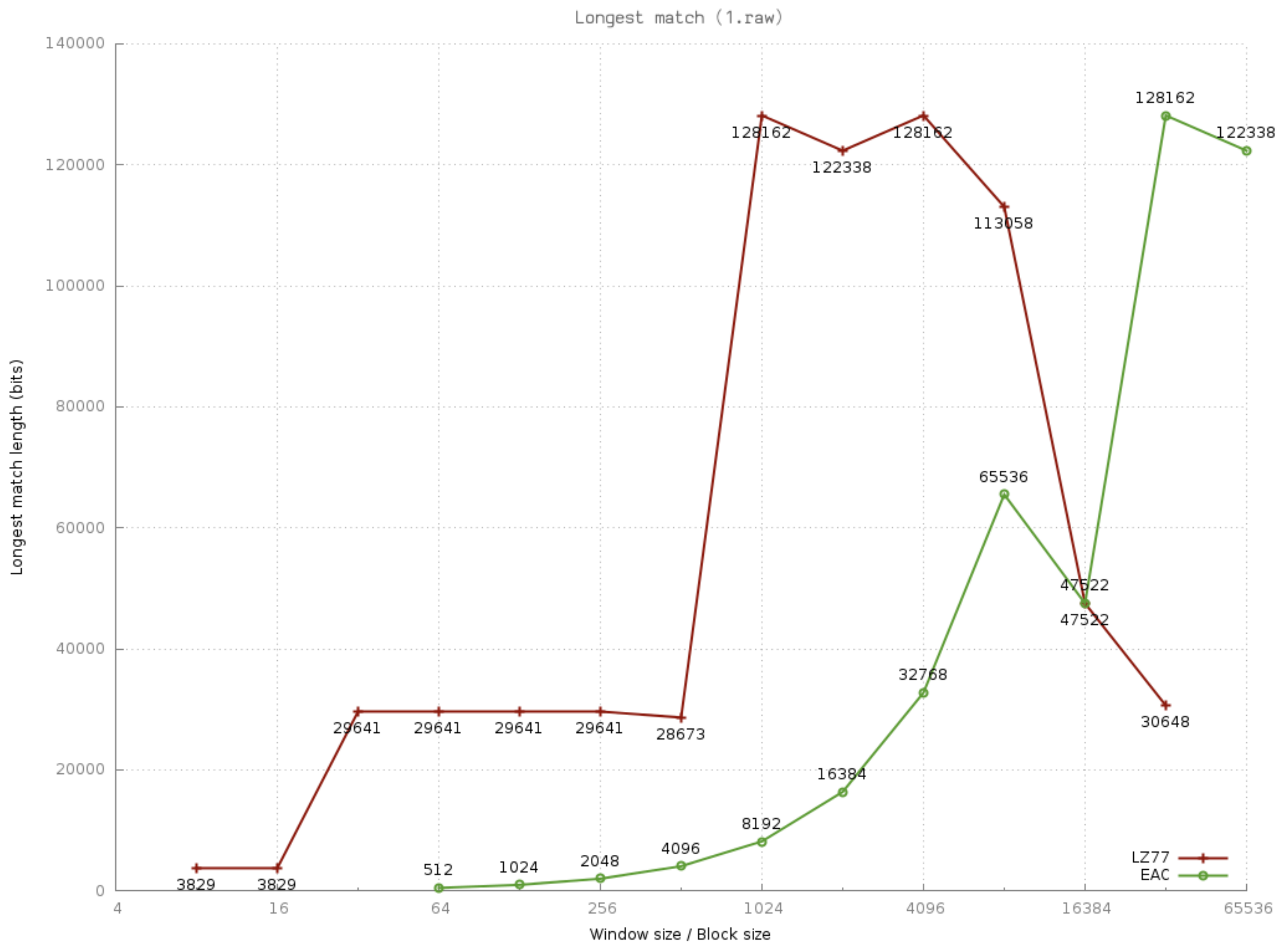
Project documentation is written using doxygen inside source code. To extract documentation and generate HTML files with documentation run command *'make doc'* in project documentation. Documentation is generated into 'doc/html' directory. To read documentation open doc/html/index.html file.

Appendix A

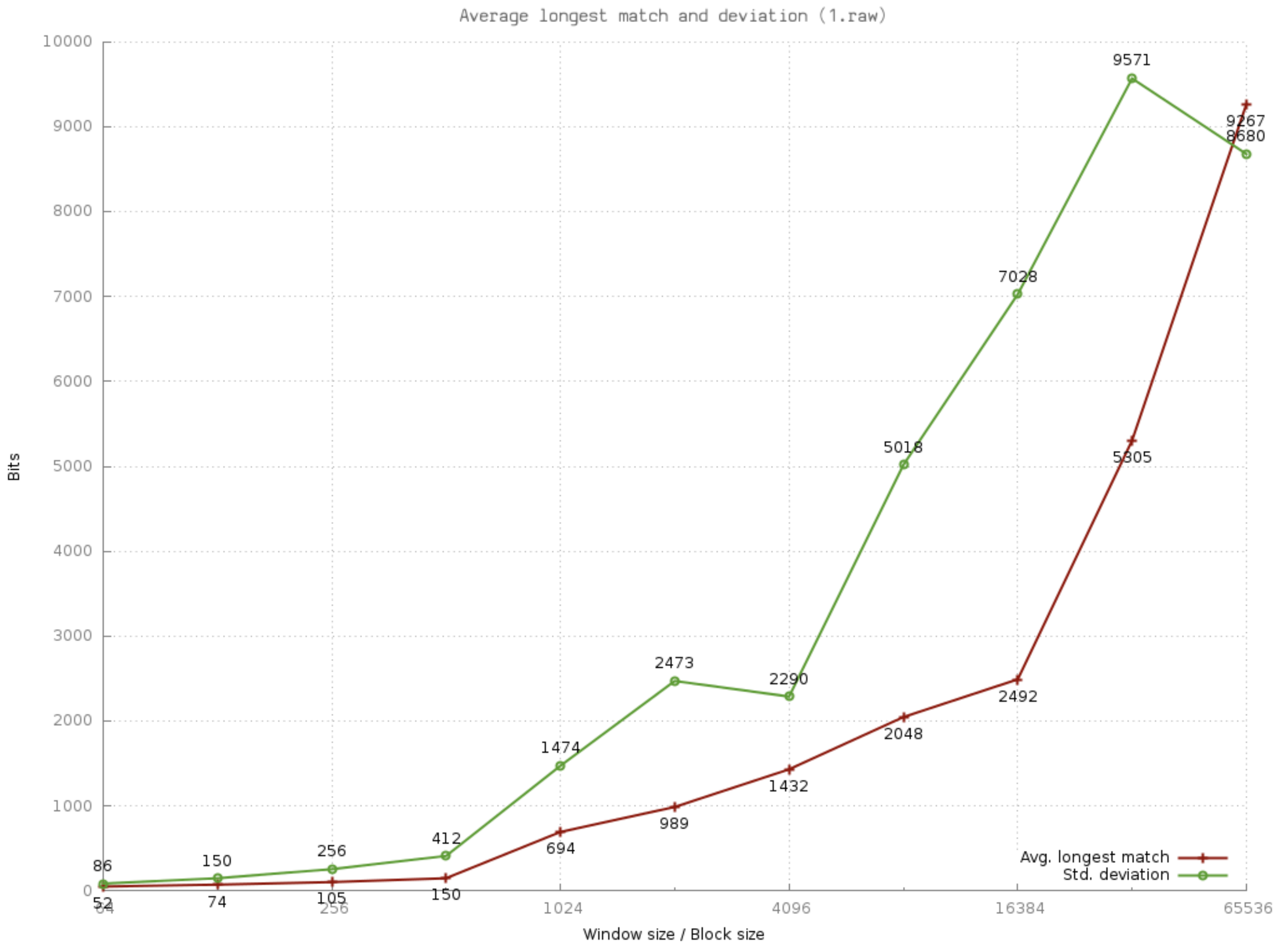
Partial list of graphs

A.1 File - 1.raw



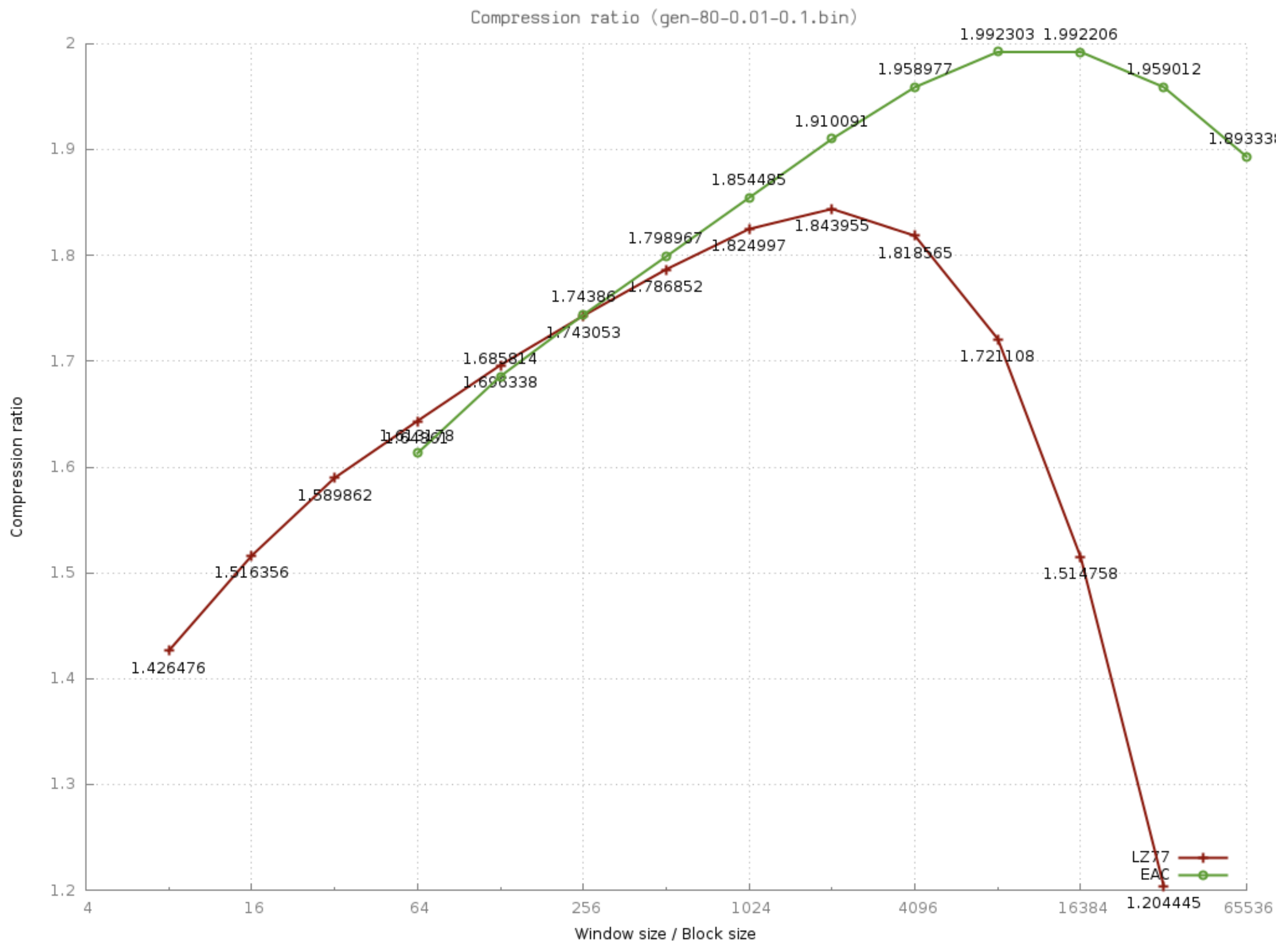


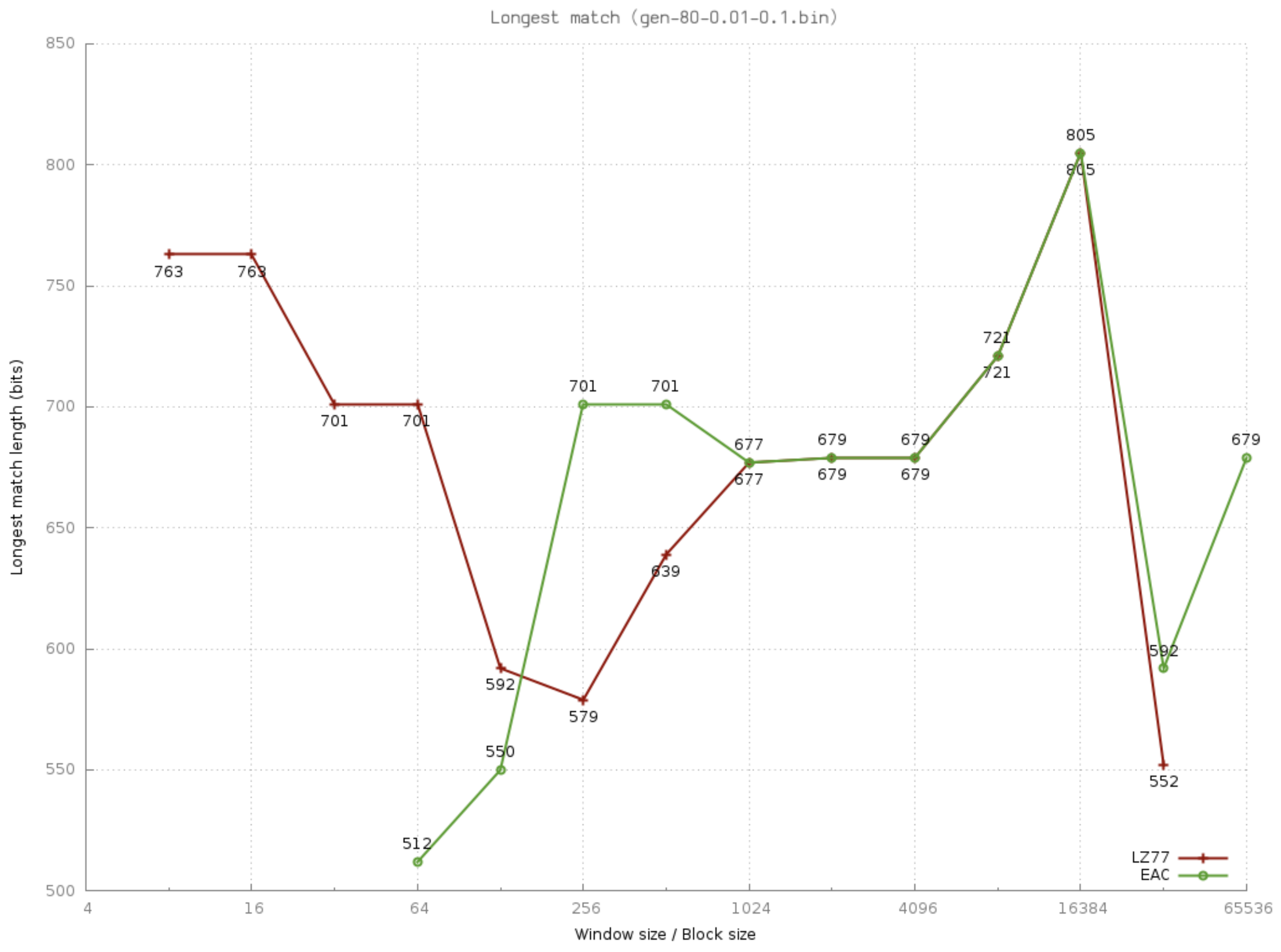
Longest match (1.raw)
Longest match length (bits)
Window size / Block size
0
20000
40000
60000
80000
100000
120000
140000
4
16
64
256
1024
4096
16384
65536
LZ77
EAC
3829
3829
29641
29641
29641
29641
28673
128162
122338
128162
113058
47522
30648
512
1024
2048
4096
8192
16384
32768
65536
47522
128162
122338

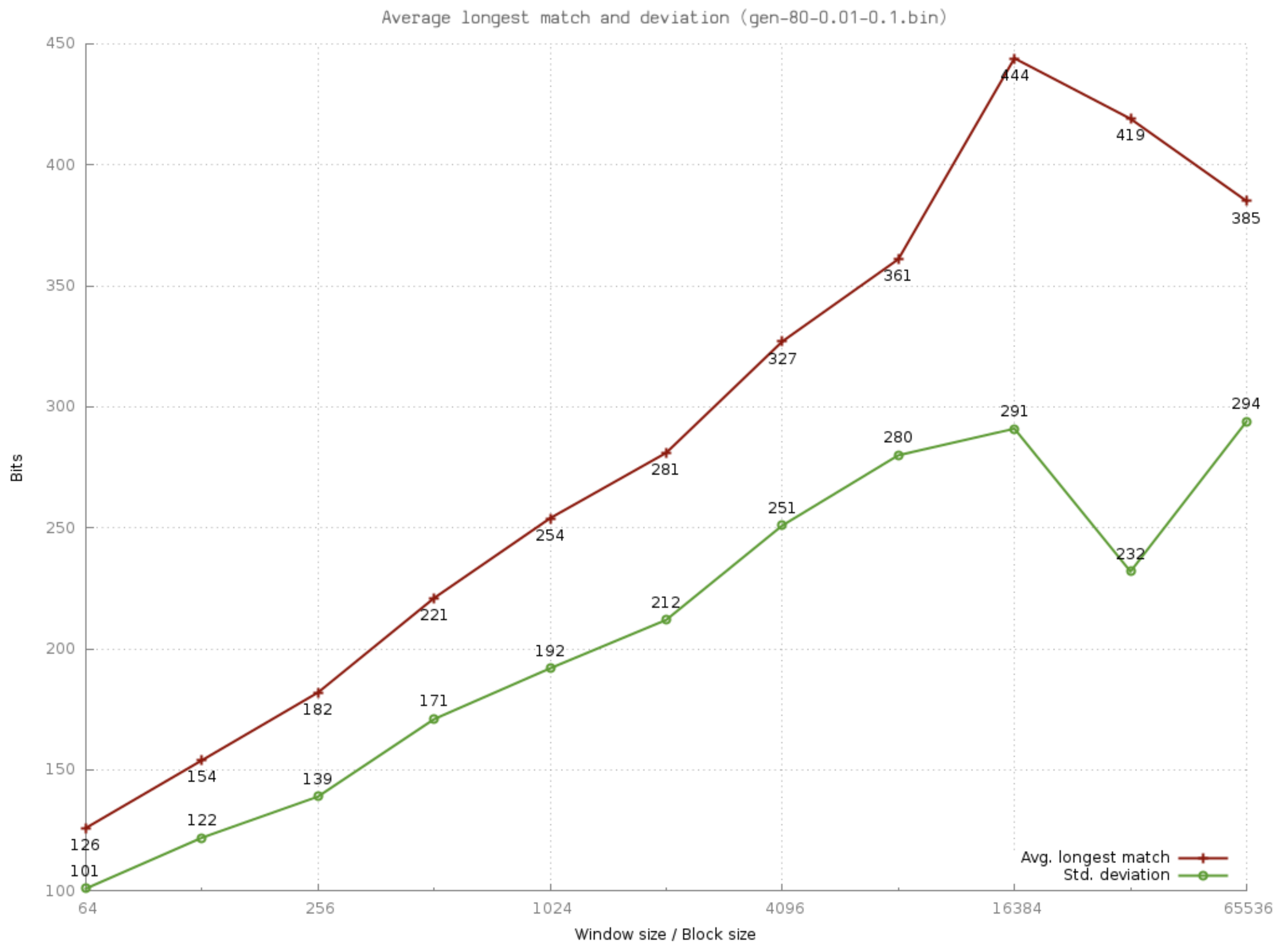


Average longest match and deviation (1.raw)
Bits
Window size / Block size
Avg. longest match
Std. deviation
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
64
256
1024
4096
16384
65536
86
150
256
412
1474
2473
2290
5018
7028
9571
8680
52
74
105
150
694
989
1432
2048
2492
5305
9267

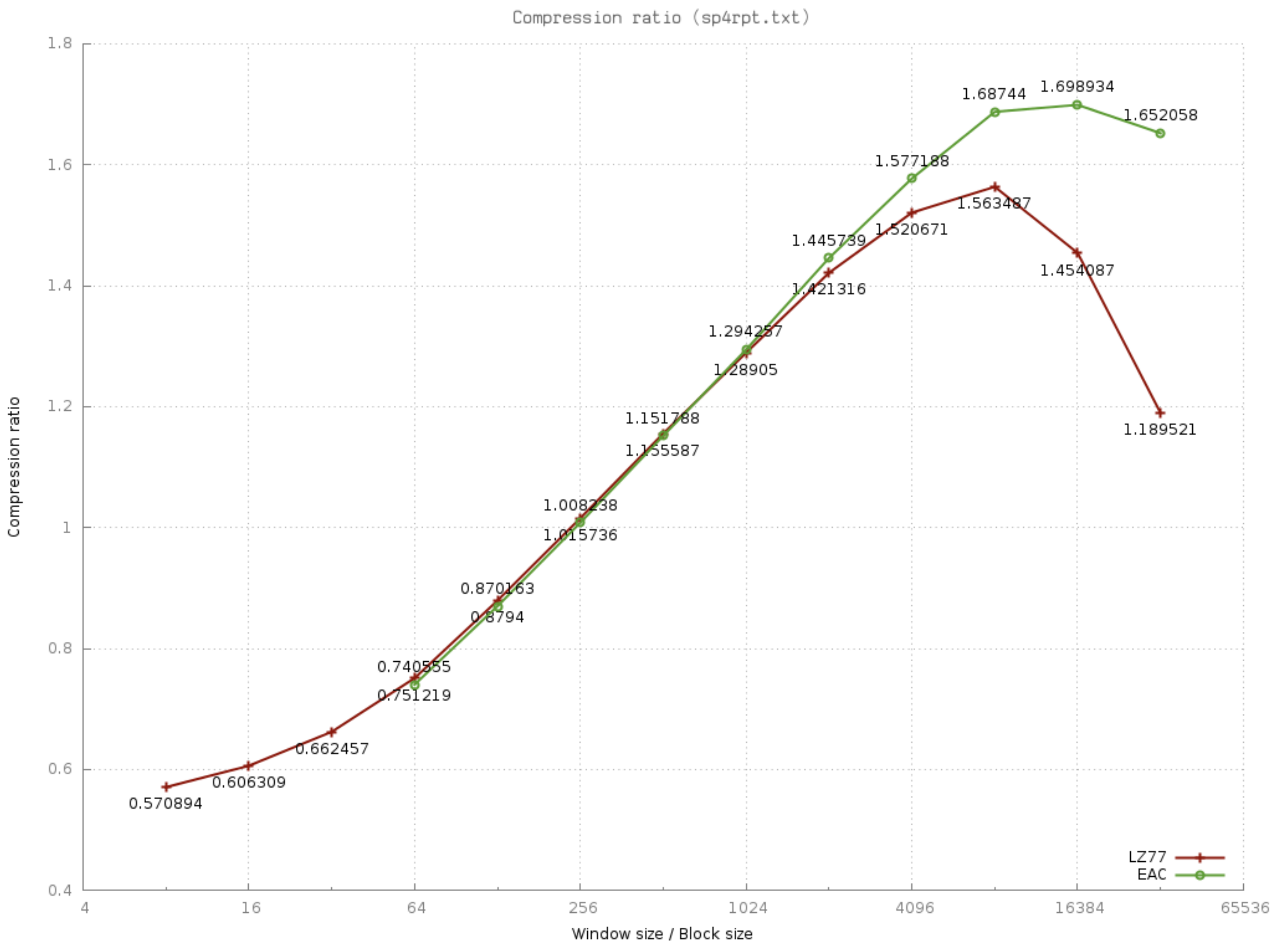
A.2 File - gen-80-0.01-0.1.bin

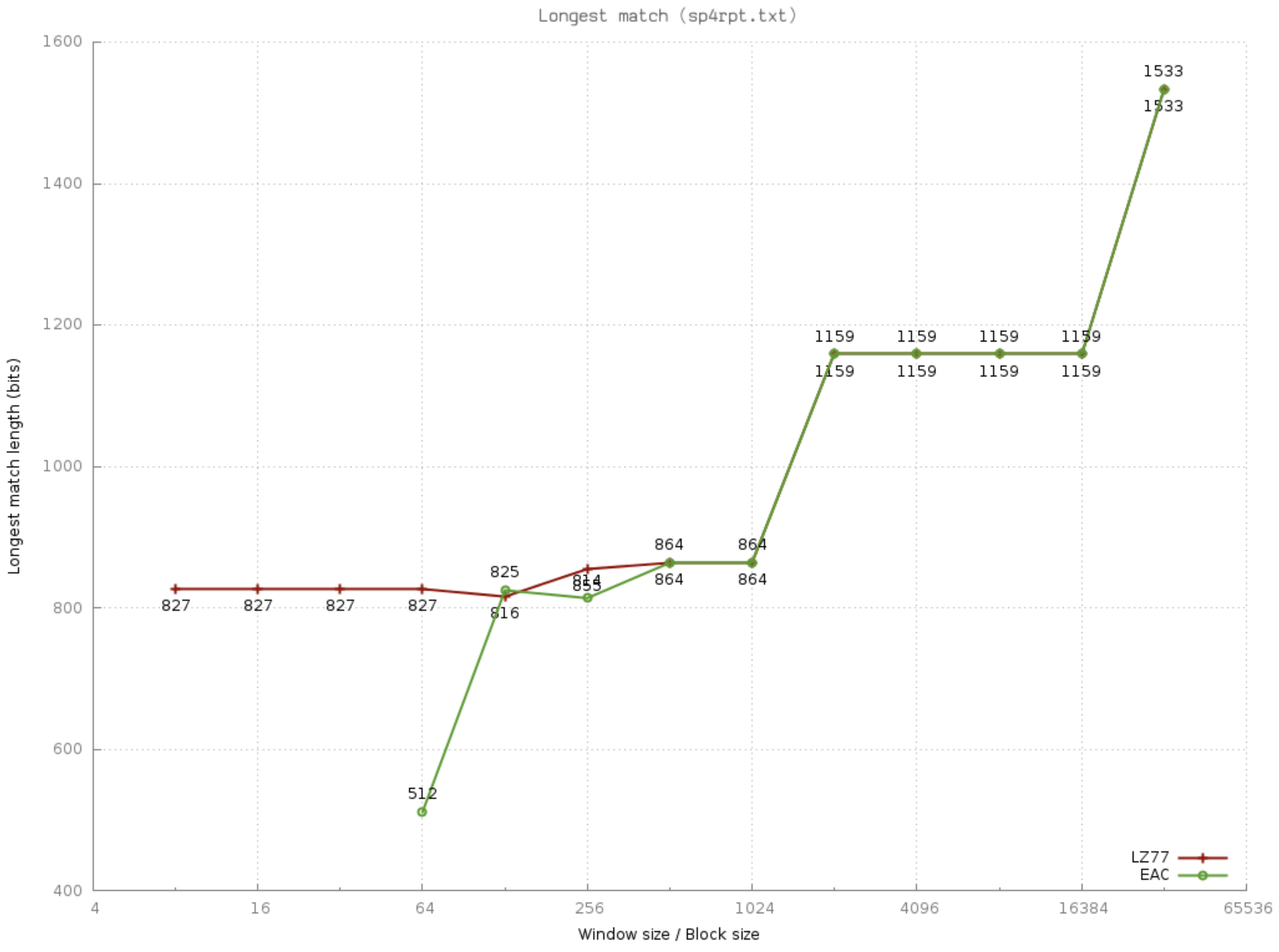



Longest match (gen-80-0.01-0.1.bin)
Longest match length (bits)
Window size / Block size
850
800
750
700
650
600
550
500
4
16
64
256
1024
4096
16384
65536
LZ77
EAC
763
763
701
701
592
579
639
677
679
679
721
805
552
512
550
701
701
677
679
679
721
805
592
679

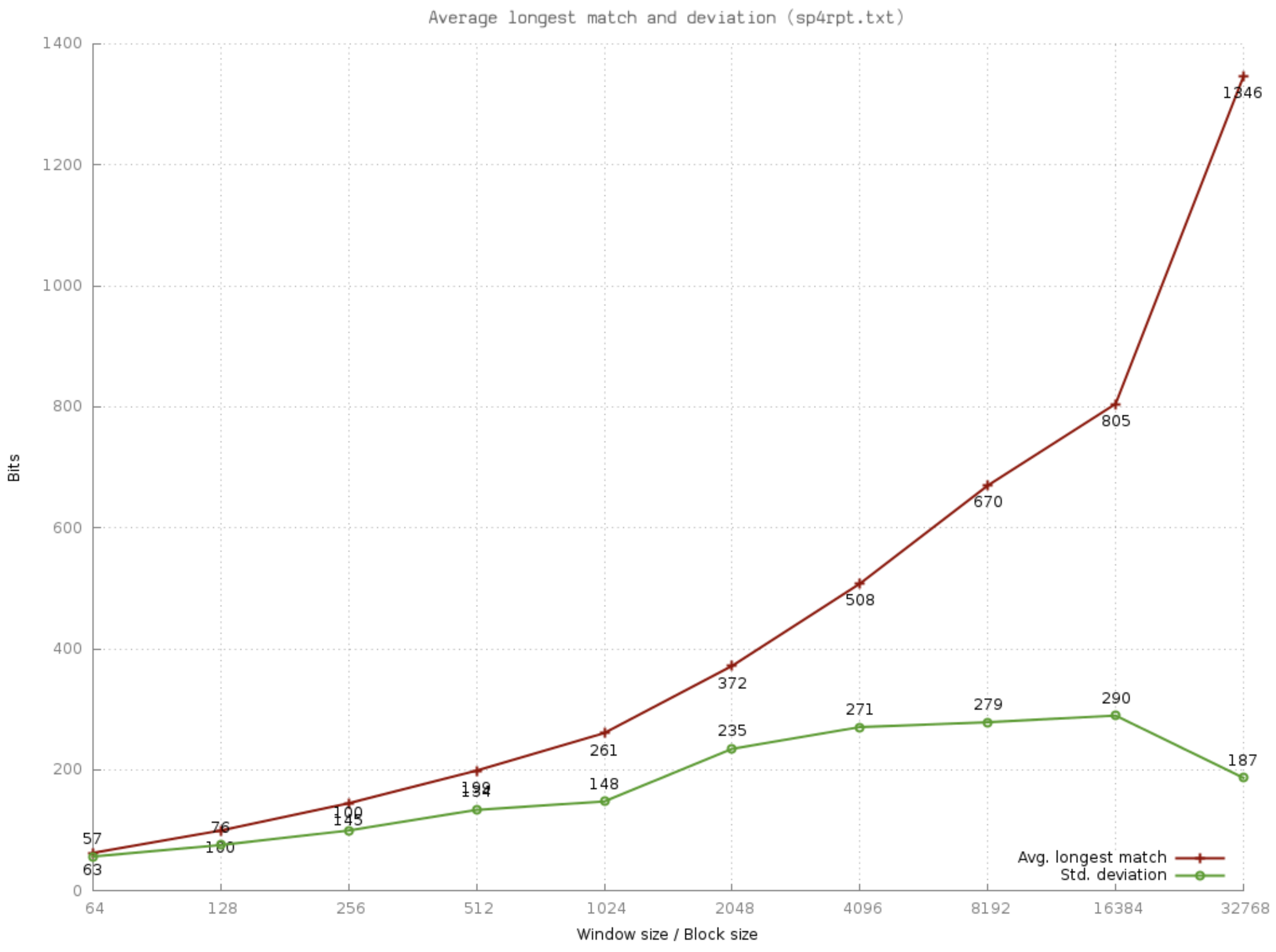

Average longest match and deviation (gen-80-0.01-0.1.bin)
Bits
Window size / Block size
450
400
350
300
250
200
150
100
64
256
1024
4096
16384
65536
Avg. longest match
Std. deviation
126
154
182
221
254
281
327
361
444
419
385
101
122
139
171
192
212
251
280
291
232
294

A.3 File - sp4rpt.txt

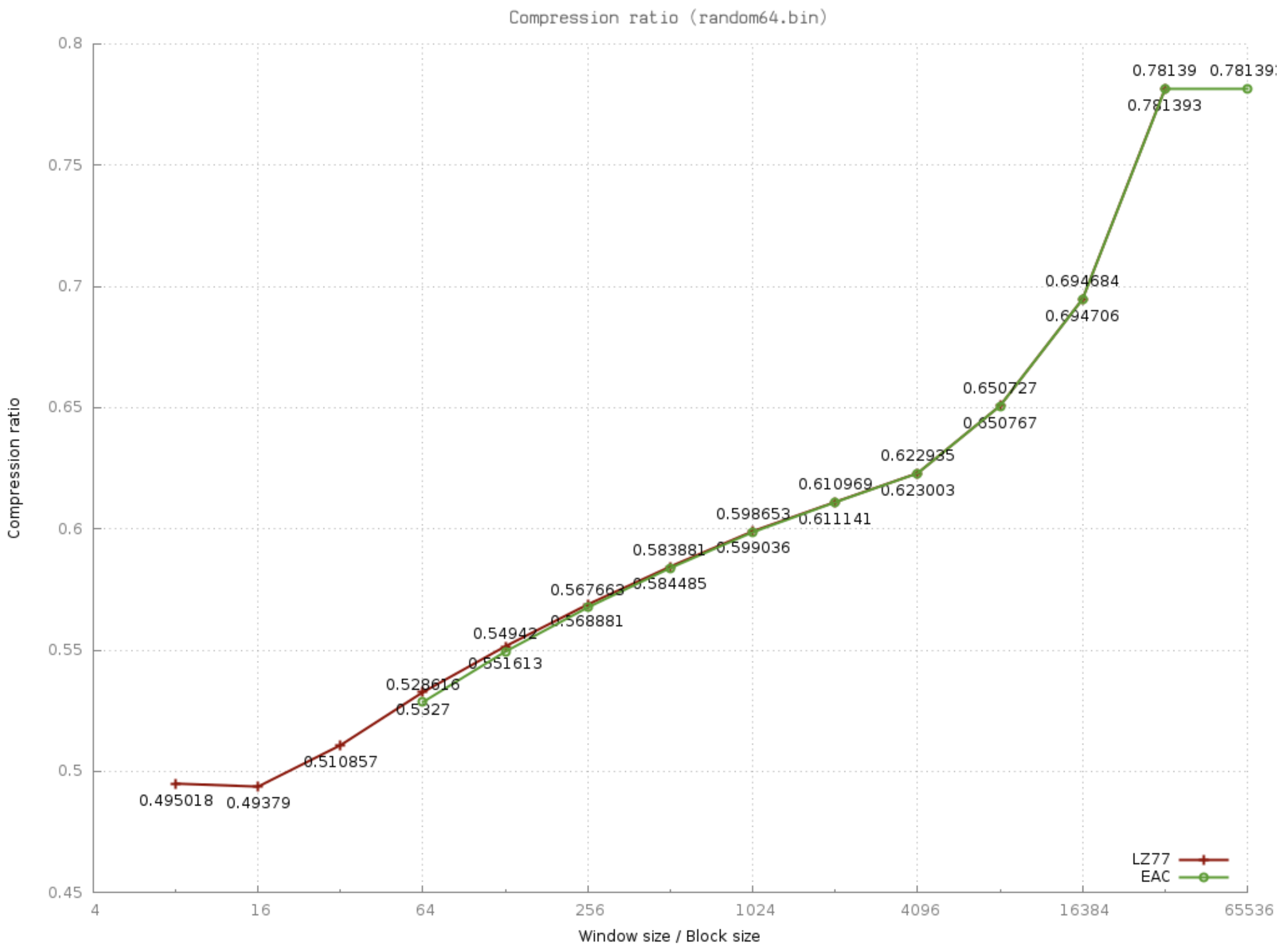


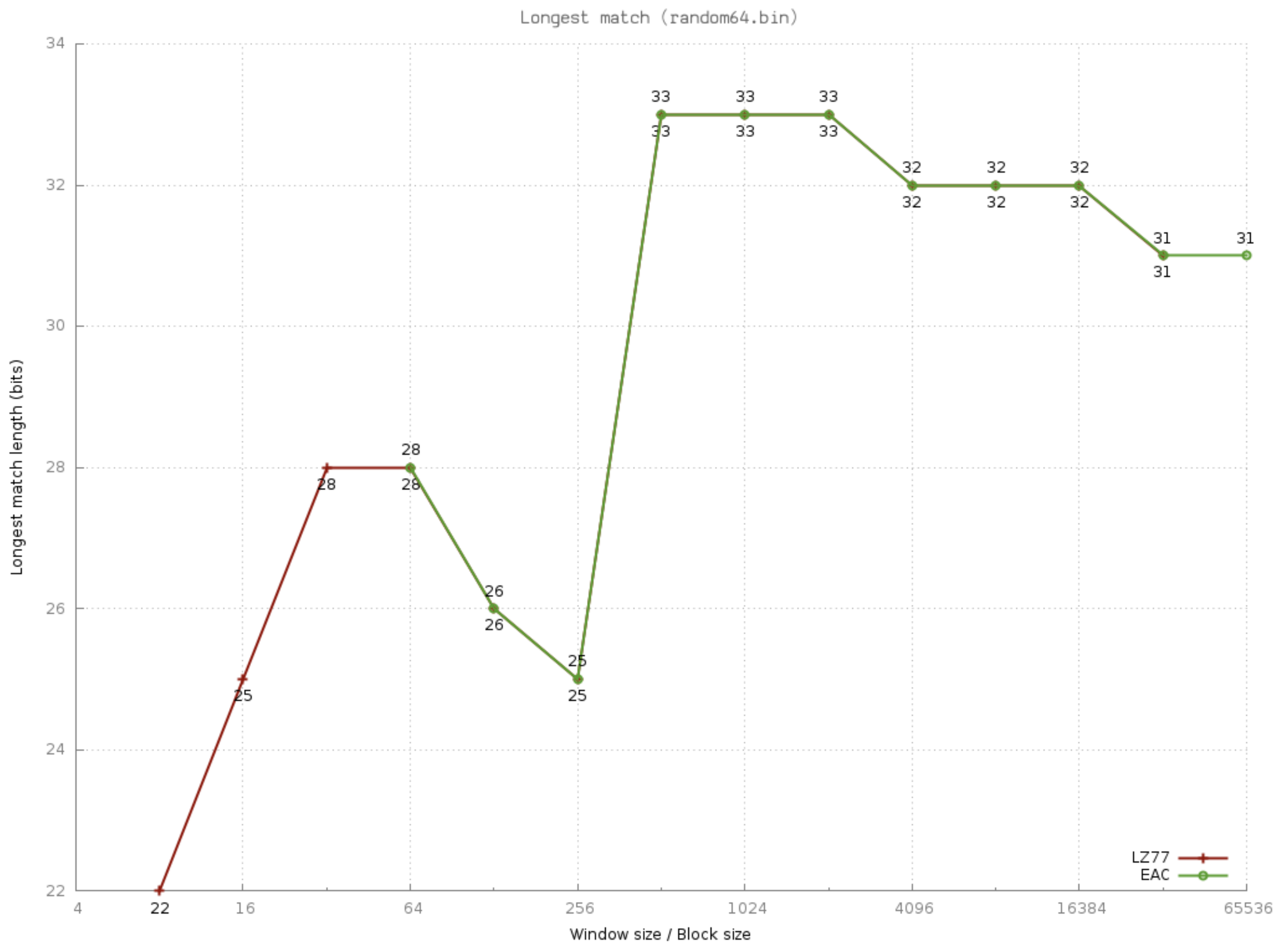


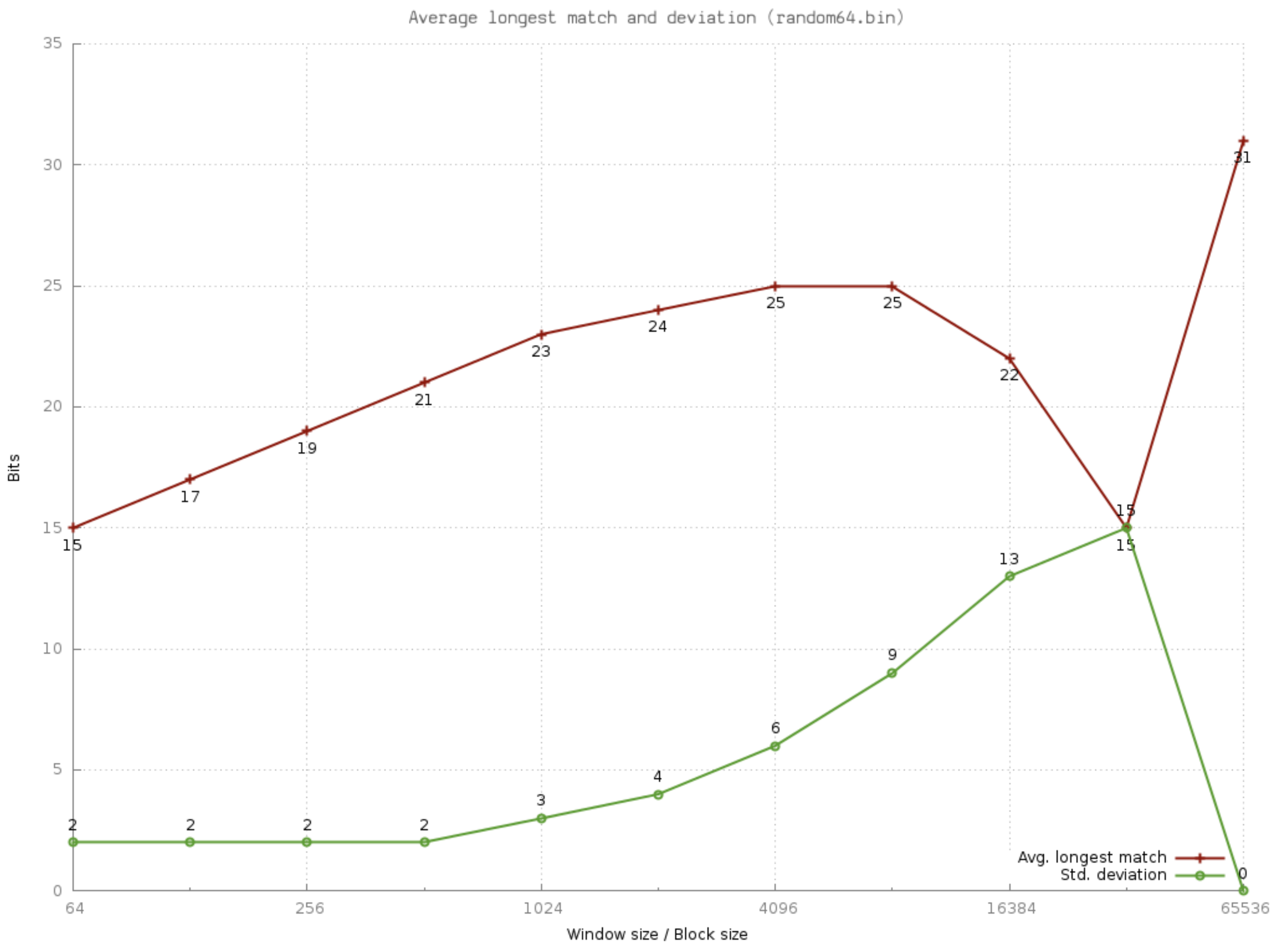
Longest match (sp4rpt.txt)
Longest match length (bits)
Window size / Block size
1600
1400
1200
1000
800
600
400
4
16
64
256
1024
4096
16384
65536
827
827
827
827
825
816
814
853
864
864
864
864
512
1159
1159
1159
1159
1159
1159
1159
1159
1533
1533
LZ77
EAC


Average longest match and deviation (sp4rpt.txt)
Bits
Window size / Block size
Avg. longest match
Std. deviation
1400
1200
1000
800
600
400
200
0
64
128
256
512
1024
2048
4096
8192
16384
32768
57
63
76
100
100
145
199
134
261
148
372
235
508
271
670
279
805
290
1346
187

A.4 File - random64.bin




Longest match (random64.bin)
Longest match length (bits)
Window size / Block size
LZ77
EAC
34
32
30
28
26
24
22
4
16
64
256
1024
4096
16384
65536
33
32
31
28
26
25
22


Average longest match and deviation (random64.bin)
Bits
Window size / Block size
Avg. longest match
Std. deviation
35
30
25
20
15
10
5
0
64
256
1024
4096
16384
65536
15
17
19
21
23
24
25
25
22
15
31
2
2
2
2
3
4
6
9
13
15
0

Bibliography

[1] Jacob Ziv and Abraham Lempel. ”A Universal Algorithm for Sequential Data Compression.” *IEEE TRANSACTIONS ON INFORMATION THEORY, 1977: 337-343.*

[2] Aaron D. Wyner and Jacob Ziv. ”The Sliding-Window Lempel-Ziv algorithm is asymptotically optimal.” *Proceedings of the IEEE, 1994: 872-877.*

[3] Shlomi Dolev, Marina Kopeetsky and Sergey Frenkel. ”Entropy Adaptive On-Line Compression”