Losslessly Compressing Radio Spectrum Related Data for Archival

DAVID VOS, University of Twente, The Netherlands

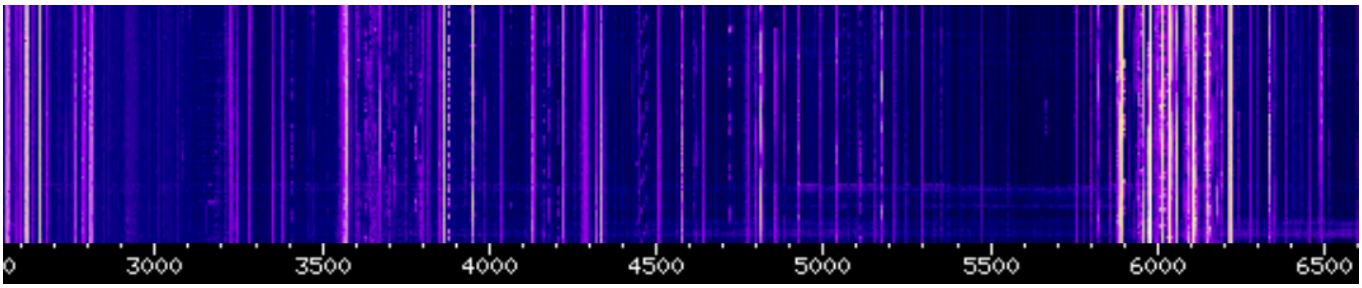

Fig. 1. A visual representation of a part of the radio spectrum

This study looks into the lossless compression of radio spectrum related data. This concerns data that is already lossy compressed. The goal of the study is to find a compression algorithm that can perform sufficiently fast and simultaneously be efficient, i.e. limit the amount of storage that the compressed data will take up. This study is part of a larger research at the University of Twente into the archival of radio spectrum data.

Bzip2, zpaq and zstd are explored in more detail in this study and are applied and benchmarked on smaller chunks of data. Within the scope of this study it can be concluded that bzip2 is most suitable for the compression of data from this type. However, there is room for further research into this area and a better algorithm could potentially be found.

Additional Key Words and Phrases: Radio spectrum, Lossless compression, Compression algorithms, Radio archival, Realtime compression

1 INTRODUCTION

As part of broader research done at the University of Twente, an approach to record and archive large amounts of received radio spectrum is investigated. Due to the nature of this data, without some compression, this data will take up vast amounts of storage space. Currently, there is already a method under investigation to apply lossy compression to the data, using insights from signal theory. This already reduces the amount of data that is being stored. After this lossy step, a lossless compression algorithm can further lower the amount of storage that the data will take up. Currently, bzip2 [15] – a general-purpose compression algorithm – is being used. However, this might not be the best fit for this kind of data, since it has a particular structure due to its nature.

The goal of this research is to compare the effectiveness of various lossless data compression algorithms on the supplied data – which has already undergone lossy compression. Besides this, research has been done into the effectiveness of changing parameters and/or tweaking parts of specific algorithms.

38th Twente Student Conference on IT, February 03, 2023, Enschede, The Netherlands

2 BACKGROUND AND GOALS

This research aims to find the most efficient lossless data compression algorithm for the specific kind of data collected by the archival of radio spectral data. Since archiving this data is done continuously, this algorithm needs to do this in real-time, otherwise, the data coming in will infinitely pile up before the compressed data can be written to storage. In short, the algorithm needs to follow these requirements:

- Perform efficiently – i.e. reduce the file size – on the type of data supplied.
- Perform fast enough to apply the compression in real-time (latency is acceptable) on the hardware used.

These requirements, especially the speed requirement, will be explained in more detail in section 4.

2.1 Data format

The data that will be used for this research will originate from a short-wave radio receiver at the ETGD[1] at the University of Twente. As an example, the receiver hardware is shown in Figure 2. This receiver receives the entire short-wave radio. The exact nature of how the data is lossy compressed is not relevant to this research, but the section below will give a short insight.

To get some insight into what the data represents, http://websdr.ewi.utwente.nl:8901/ can be used to view the radio spectrum and to listen to specific frequencies, using different radio technologies like *A.M.* or *F.M.*. Figure 1 shows a visual representation of the

[1] Experimentele Telecommunicatie Groep Drienerlo (https://etgd.utwente.nl/)

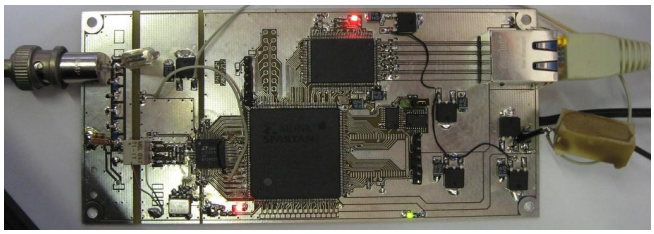

Fig. 2. An advanced radio spectrum receiver setup

spectrum, taken from this website. The horizontal axis represents the frequency and the vertical axis represents time – with the latest signal on the bottom, the colour represents the amplitude of the signal, ranging from black (no signal at all), to blue/purple (some signal), to white (high signal).

This data is recorded as a list of amplitudes for each "block" of radio spectrum data. Each block represents the amplitude of the radio signal over a width of 6.95 Hz over 0.126 seconds. For every such block, two bytes are used. The reason for this is that each block is in fact a complex number, so a real and an imaginary component of the number is stored. The reasons for this are not relevant to this research, only the fact that this is the way that the data is represented is applicable. In short, there is a list of bytes for each "line" of data – similar to a line of pixels in Figure 1. A file used for our research will repeat a large number of such lines and such a file needs to be losslessly encrypted.

It can already be noted that this data has large gaps with little to no signal, this is also apparent in the lossy compressed data, in between the chunks of small non-zero integers – i.e. the waves that can be seen in Figure 1 – there are large amounts of bytes with value zero. This will probably lend itself well to compression. Another notable point about the data is that this pattern repeats similarly in every line. After all, if there is a signal that is broadcast to a specific frequency, a value will exist on that frequency each timeframe.

2.2 Research questions

The goals of the research outlined above lead to the following research question:

What lossless data compression algorithm is most suitable and efficient for compressing radio spectrum-related data in real time?

In order to answer this question, the following sub-questions are set out:

(1) What potentially suitable lossless data compression algorithms exist?
(2) How do these selected algorithms perform on our data?
(3) In what way can (one of) these algorithms be tweaked in order to improve performance?

By answering these questions and therefore the main research question, an algorithm can be selected as being the most suitable for the set-out goal. It might be that the solution that is used currently is already the fastest or that an alternative is only marginally faster. In such a case it would not be beneficial to switch, since two types of archived data will exist without a noticeable advantage. In the case that another algorithm *is* significantly more efficient, it will be recommended to switch to that.

3 RELATED WORK

The first step in finding related work for this research is of course to narrow down the area that is related to this topic. At first, it could be easy to start looking into the compression of radio waves. However, this does not apply here. After all, the data that needs to be compressed has already been through lossy compression and is now just a series of bytes. Therefore, we start looking for work related to general lossless compression.

3.1 Lossless compression

In previous sections of this proposal, the terms *lossy* and *lossless* compression have already been used. The difference between these two methods is that lossy compression methods achieve better compression by losing some information [12]. This is mainly used for images, movies or sounds since humans cannot tell the difference if the loss of information is small.

Lossless compression, on the other hand, involves no loss of information, the original data can be recovered exactly from the compressed data [13].

3.2 General literature

There exist some literature sources that cover a wide variety of different types of compression and can be used as a starting point for finding related work. Examples are books by Salomon and Sayood. Both books cover lossy and lossless compression and go into thorough detail on different compression algorithms for a wide variety of use cases. Not relevant to this research are the sections on lossy compression and general compression of unrelated data like images or audio. Such literature can mainly be useful when looking into why certain algorithms perform better or worse on certain data (see also section 8).

3.3 Compression algorithms

While various general literature is important in this research, in order to find suitable candidate algorithms for the research, sources outside scientific literature are also vital. This is because algorithms or implementations of compression algorithms might have been developed but not scientifically studied or published.

Most of these algorithms have extensive documentation that can be consulted in order to learn about their features and underlying technologies. For example, bzip2 – which is currently used in the wider research – has a large and extensive manual where not only the usage of the program is documented, but also the various parameters that can be customised as well as some insight into the data structure [15].

4 METHODOLOGY

In order to find the most efficient algorithm for the use case of this research, the methodology described in this section has been applied. The steps described here are in line with the sub-research questions described in subsection 2.2.

4.1 Selecting algorithms

The first step is to find a set of compression algorithms that might be suitable for the purposes of this research. For this, the literature described in section 3 has been consulted. Furthermore, a search has been conducted on some non-academic platforms.

To start, a Google search was conducted using the following keywords: *lossless compression algorithms, bzip2 algorithm, data compression, lossless compression, ppm family compression, lossless file compression, fast lossless file compression.* Some of these terms have their origin in results from earlier search queries. For example, after searching for lossless compression, the results lead to *gzip*[5], where the PPM family of compression is mentioned.

Besides this, open-source compression projects often provide links to similar projects. Furthermore, the Wikipedia pages of some projects also link towards the family of compression that a certain algorithm belongs to and to related projects.

4.2 Initial benchmarking

Measurements have been taken to find out how effective a particular algorithm is on the data set for this research. This was done in a few steps.

(1) Run all algorithms on some example data.
Some example files have been collected during various times of the day and from various portions of the radio spectrum. These files vary in their type of content because certain portions of the spectrum get different kinds of activity and the time of day also influences what is being broadcast. Therefore, a wide variety of test files were provided. The full list can be found in Appendix A.
Initially, file 54 was chosen because it contains a lot of variety, i.e. there are large parts with almost no signals but also some strong broadcasters.
A simple Python script has been written in order to record the time taken and compressed file size produced by each algorithm.

(2) Tweaking the algorithms.
In order to select the best-performing algorithm, the study also looked at a way to tweak existing algorithms. This can be done by changing the parameters given to the programs. The variable parameters of the algorithms were inspected and the benchmarks described in the previous subsection have been run with various variations of these parameters.
The algorithms with tweaked algorithms were run on a larger part of the test files.

4.3 Benchmarking smaller chunks

The first two steps described above use the command line implementation of the compression algorithms in order to compress full files. In the context where the algorithm will be used, this is not the case. After all, the data will come in from the radio receiver in real time. Therefore small chunks of data will need to be compressed at a time, rather than one large file. To benchmark such a situation, a program was written in C++ that will look through a file in chunks of a few lines[2] and call the algorithm to compress each chunk and write it to a file. This was done using the libraries provided by the compression algorithms.

5 RESULTS

5.1 Algorithms

The following list of algorithms was selected. This research will not go into detail about the various underlying algorithms used by these compression algorithms, but they are listed here to serve as a comparison and reference.

- **Gzip**: A popular data compression program written for the GNU project [5]. It makes use of the Lempel–Ziv coding (LZ77) algorithm and Huffman coding [8]. The combination of L77 and Huffman used in Gzip is also known and specified as DEFLATE [2].
- **Bzip2**: "A freely available, patent-free, high-quality data compressor." [14]. This algorithm is also included by default in many Linux distributions and was designed to use a similar syntax to Gzip [14]. It is what the wider research currently uses. It uses the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding [15].
- **Zopfli**: A compression algorithm written by Google that can perform DEFLATE, or zlib compression. It achieves better compression than the normal implementations of these algorithms but at the cost of being slower [6].
- **Zstandard**: "A fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios. It's backed by a very fast entropy stage, provided by Huff0 and FSE library." [3] It also offers a dictionary mode where a dictionary can be trained on a large amount of data from a dataset, after which small amount of similar data can be compressed more effectively and faster [3].
- **zpaq**: an incremental, journaling archiver. "For backups it adds only files whose date has changed, and keeps both old and new versions. You can roll back the archive date to restore from old versions of the archive." [17]. This is not very useful for our purposes, however, it also promises faster compression times and better compression ratios [11].
- **XZ**: a general-purpose data compression algorithm with a high compression ratio, using the LZMA2 algorithm. "With typical files, XZ Utils create 30 % smaller output than gzip and 15 % smaller output than bzip2." [16]
- **LZ4**: an algorithm aiming to provide a balance between speed and compression ratio [9]. It uses the LZ77 algorithm, but does not combine it with an entropy coding stage, like in Huffman in DEFLATE [10]. It can also use Zstandard dictionaries in order to perform better on smaller amounts of data from a larger dataset [9].

5.2 Initial benchmarks

5.2.1 Default settings. For the initial benchmarks, each algorithm was run with its default settings on file 54, as explained in the methodology. The results can be found in Figure 3. This figure shows the time it took the algorithm to compress the file and the storage size of the compressed file, relative to the original filesize.

The benchmarks were run with a maximum time of 600 seconds. Both zopfli and xz did not finish within this time and therefore are too slow for the purposes of this research, at least with the default settings. This does however not mean that they will be excluded from the next steps, with tweaked parameters they might perform faster.

[2] a line as described in subsection 2.1, ergo a chunk will be a few kilobytes in size.

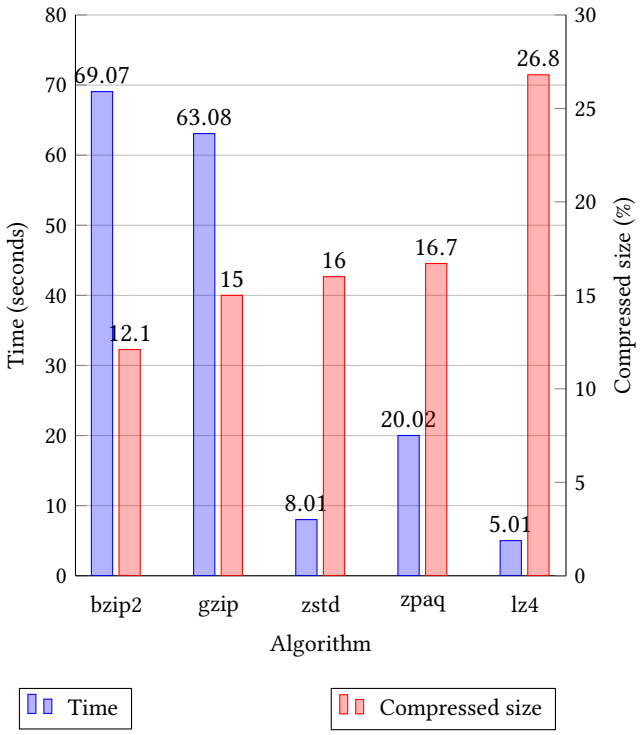

Fig. 3. Initial benchmark results

5.2.2 Command line parameters. In order to further benchmark the algorithms on the provided data, the parameters for each algorithm were inspected. The parameters that are useful for this research (mainly the parameters controlling the compression rate) were selected. They can be found in Table 1.

Table 1. Command line parameters of the algorithms

Name	Options	Description
bzip2	-s –small	reduce memory usage but also speed
	-1 to -9	sets block size, does not really affect speed
gzip	-1 to -9	-1 is fasted but least compression, -9 is slowest but most compression, default = 6
zopfli	–i#	# is the amount of iterations, more gives higher compression but is slower, default is 15
zstd	-1 to -19	compression level, faster to better, default = 3
	–ultra	enable compression levels up to 22, requires more memory
	-D DICT	use DICT as dictionary (might be interesting for small amounts of data)
	–train ##	create a dictionary from a set of training files
zpaq	-m0 to -m5	compression level, faster to better, default = 1
xz	-0 to -9	compression level, faster to better, default = 6
	-e	try to improve compression ratio by using more CPU time
lz4	-1 to -9	compression level, faster to better, default = 1

A range of options for these parameters have been chosen and they have been run on a random selection of the files. The selection of commands that were run can be found in Appendix B. The most interesting and useful results are shown in Figure 4.

Some remarks about the algorithms left out from the figure:

- LZ4 has been excluded since it did not produce comparably high compression rates to bzip2, even at its highest settings.
- Zopfli has been excluded since it failed to finish in under 600 seconds.
- XZ has not been included. Even though the data from Appendix C would make you believe that it is very efficient, it only managed to finish in time for a few of the files and was too slow for the rest.
- Zstd on compression level 22 (ultra) performed even more efficiently (most notably so in the best cases, and therefore also on average) and also slower, but has been left out from this graph to keep the graph easy to read.

N.B. the scales on this graph are different than in Figure 3 in order to fit these results better. After all, the algorithms shown here are set to higher levels than their defaults and therefore finish in more time and produce lower compressed sizes.

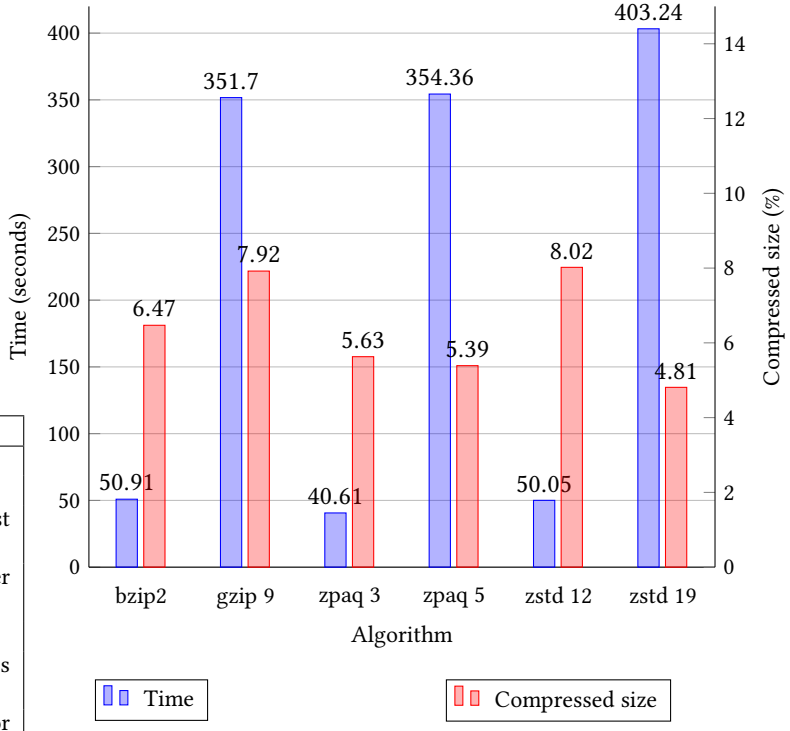

Fig. 4. Average results of benchmarks with tweaked parameters

It can be seen that in this scenario, only bzip2 and zpaq are close in terms of time. Figure 4 only shows the average time and ratio. From looking at Appendix C it can also be seen that zpaq outperforms bzip2 in some cases. Hence, they have been chosen to further inspect for the next part of the research. Furthermore, since processing such small amounts of data at a time might lead to worse

Table 2. Chunk sized benchmarks per file

	0	2	12	15	18	20	22	27	28	30	40	44	46	47	52	56	59	61
bzip2	6.40%	19.54%	3.58%	6.69%	3.81%	5.18%	4.78%	11.25%	3.34%	3.09%	9.25%	11.43%	23.23%	19.63%	3.16%	4.43%	6.73%	5.38%
zstd w/ dict	6.70%		3.90%	7.24%	3.96%	5.39%	5.33%	11.45%	3.46%	3.34%	9.46%	12.45%	24.80%		3.66%	4.69%	6.94%	5.56%
zstd w/o dict	6.69%		3.90%	7.24%	3.95%	5.39%	5.32%	11.45%	3.45%	3.34%	9.47%	12.45%	24.82%		3.65%	4.69%	6.94%	5.55%
zpaq	7.23%	19.12%	4.33%	7.43%	4.57%	6.04%	0.11%	11.40%	4.31%	4.07%	9.56%	11.65%	22.54%	19.21%	0.02%	5.24%	7.64%	6.31%

Table 3. Chunk sized benchmarks, average

algorithm	average size	average time
bzip2	6.984%	74.6
zstd with dictionary	7.395%	377.6
zstd without dictionary	7.395%	392.3
zpaq	7.028%	227.7

compression ratios, zstd has also been chosen as a candidate because of its dictionary option, as outlined in subsection 5.1.

5.3 Smaller chunks of data

To benchmark how the algorithms perform on smaller chunks of data, a C++ program has been written. As described in subsection 4.3, the program uses the libraries of the algorithms on chunks of spectrum data. Table 2 shows the results for each algorithm and file. A colour scale was applied to indicate compression ratio. Table 3 shows each algorithm's average compression size and time. It should be noted that this average excludes files 2 and 47 in order to create a fair comparison with zstd since it failed to compress these files in under 600 seconds.

The following parameters were used for these benchmarks:

- bzip2: its default settings, since it does not expose any settings regarding compression level.
- zpaq: compression level 3, from the results of the previous benchmarks it is clear that increasing this level does not contribute significantly to the efficiency but mainly increases time.
- zstd: compression level 19, from the previous benchmarks this level shows to provide very efficient results, albeit slower than bzip2 and zpaq. Higher levels are also possible, but for the sake of research speed, level 19 was settled on.
  It was run both with and without a dictionary. This dictionary was trained on the first 131072 bytes of each file (the default behaviour of zstd training).

N.B. zstd and zpaq used some multithreading by default in the first results (subsection 5.2). In the C++ implementation of this benchmark, this was not the case. Therefore, the results in Figure 3 and Figure 4 are not directly comparable to those in Table 3.

6 DISCUSSION

There are a few things that can be noted about the results of this research.

- After the benchmarks on individual files, zpaq and zstd seemed to be good candidates for the goals of this research. They performed slower but with smaller compressed files.
- When benchmarking on smaller amounts of data – similarly to how the data will be processed in the wider research – these improvements over bzip2 do not seem to be present anymore.
  This can be explained by two factors:
  (1) The algorithms apply compression techniques that are more efficient when larger amounts of data are being compressed. They might for example be able to compress a set of data better in the context of more similar data.
  (2) The algorithms use multithreading techniques when benchmarking on complete files, but not when implementing their libraries to work on smaller amounts of data. This is true for both zstd and zpaq and can be seen when inspecting the processes that these programs spawn (4 and 10 respectively). For the conclusions of this research, we care about single-threaded performance. In the context of the wider research the compression already gets performed in parallel so this is not something that needs to be benchmarked in this research. Besides, for a fair comparison between the algorithms, they should be benchmarked in the same context, i.e. single-threaded.
- Point 1 can potentially be solved by the dictionary mode that zstd provides. This dictionary is trained on a set of data and can then be applied when training so that the algorithm can use more of this context. This is explained in more detail in the zstd documentation [4].
- In Table 2, it can be seen that files 2 and 47 are compressed worse than most others by bzip2 and zpaq, and zstd is not able to finish compressing it in under 600 seconds. These files are larger than most others (2.2GiB and 1.7GiB relatively) and are both in the 480-1650 kHz range. In Europe, the medium wave broadcasting range (on which licensed commercial stations can broadcast) is from 526.5-1606.5 kHz [1]. This would explain the large amounts of data in this range and could also explain this being harder to compress.
- Table 2 also shows that file 46 was significantly harder to compress. This file cointains the range from 1-300 kHz. This is what is known as longwave radio [7]. On this range there is a lot of signal activity like radionavigation as well as longwave broadcasts [1]. This probably explains why this file is harder to compress; there exists a lot of data here, as opposed to other ranges which had a lot of empty space.
- Then, there are files 22 and 52, on which zpaq performs extraordinarily well. These files are both from the 12000-12905 kHz range. An explanation as to why zpaq performs so well on these files has not been found and can be left as future work.

7 CONCLUSION

With the discussed items in section 6 in mind, we can form a conclusion about the results of this research.

Linked to the research questions set out we can say the following:

(1) There exist numerous compression algorithms that are potentially suitable for the compression of radio spectrum-related data.
(2) After benchmarking these algorithms on our dataset, the most suitable candidates proved to be bzip2, zpaq and zstd. After testing these algorithms on smaller amounts of data – somewhat similar to their application in the wider research project – bzip2 proved to be both the fastest and most efficient on this kind of data, overall. In some cases, zpaq performed better. More research can be done into this.
(3) Most algorithms have some way of tweaking their compression level. Zstd also has a way to train the algorithm to behave more efficiently on a specific dataset.

Taking this into account, this paper arrives at the following conclusion: bzip2 is the most suitable and efficient algorithm for compressing the type of data used in this study. Zstd and zpaq can come close to the rate compression of bzip2 but do not succeed to be better and also do so in significantly more time – when looking at the average results. Having said that, there is significant room for future work which might arrive at the conclusion that another algorithm, with the correct configuration, is more suitable.

8 FUTURE WORK

There are some things that are unexplored yet in this research.

Firstly, only bzip2, zpaq and zstd were benchmarked on small chunks of data. Further research could implement and benchmark more algorithms in this way. This study has limited itself to doing so for only these three most promising algorithms due to time constraints. Additionaly, xz has shown to be able to produce very high compression ratios, albeit in a significantly large amount of time. Maybe this algorithm can be applied in such a way that it performs quicker on this set of data. Or maybe the larger time is a tradeoff that might prove to be worth it. In any case, it has not been sufficiently researched in this study to form any final conclusions about it.

Likewise, the size of the chunks could also play a role in the performance of the algorithm. This was made a variable in this study but has not been explored thoroughly.

Moreover, the training mode of zstd was not explored deeply. The zstd documentation promises much more drastic results than were evident in this study [4]. It might be that by selecting the training data more carefully, the performance of the algorithm will be improved. In this research only a relatively small amount of training data was used, as per the default settings of zstd. However, the program also provides a way to split files into more training samples. This might prove useful in order to gather better results.

Furthermore, research can be done into pre-processing the data in some way. One suggestion might be to transpose each chunk of data so that signals on the same frequency follow each other – instead of signals from the same timeframe. This might improve efficiency since this might produce a pattern that an algorithm can process more efficiently.

More research could also been done into why certain algorithms perform well on certain files. For example, zpaq performed very well on some specific files. An explanation might be found for this that can lead into more insights regarding the efficient processing of this data.

Lastly, a custom version of an algorithm could be created by modifying its source code and writing it such that it is designed to work very efficiently on this specific type of date. An algorithm could even be written from scratch.

REFERENCES

[1] Electronic Communications Committee et al. 2013. The European table of frequency allocations and applications in the frequency range 8.3 kHz to 3000 GHz (ECA table). In *Proceedings of European Conference of Postal and Telecommunications Administrations; Electronic Communications Committee: Copenhagen, Denmark.*
[2] L. Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. https://doi.org/10.17487/RFC1951
[3] Facebook. 2022. zstd - README. https://github.com/facebook/zstd
[4] Facebook. 2022. zstd - The case for Small Data compression. https://github.com/facebook/zstd#the-case-for-small-data-compression
[5] Free Software Foundation, Inc. 2022. GNU Gzip. https://www.gnu.org/software/gzip
[6] Google. 2016. Zopfli - README. https://github.com/google/zopfli
[7] R.F. Graf. 1999. *Modern Dictionary of Electronics.* Elsevier Science, 23. https://books.google.nl/books?id=AYEKAQAAQBAJ
[8] Free Software Foundation, Inc Jean-loup Gailly. 2022. GNU Gzip. https://www.gnu.org/software/gzip/manual/gzip.html
[9] lz4. 2020. lz4 - README. https://github.com/lz4/lz4
[10] lz4. 2022. LZ4 Block Format Description. https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md
[11] Matt Mahoney. 2016. ZPAQ - Incremental Journaling Backup Utility and Archiver. http://mattmahoney.net/dc/zpaq.html
[12] David Salomon. 2004. *Data Compression: the complete reference.* Springer Science & Business Media, London, England, UK.
[13] Khalid Sayood. 2017. *Introduction to data compression.* Morgan Kaufmann.
[14] Julian Seward. 2019. bzip2 and libbzip2. https://sourceware.org/bzip2/index.html
[15] Julian Seward. 2019. bzip2 and libbzip2, version 1.0.8. https://sourceware.org/bzip2/manual/manual.html
[16] Tukaani Developers. 2023. XZ Utils. https://tukaani.org/xz
[17] zpaq. 2016. zpaq - README. https://github.com/zpaq/zpaq

9 APPENDICES

A TEST DATA

#	Time	Bands (kHz)	Filesize
0	00:15	3900-4805	1.7GiB
1	00:45	1-300	586.7MiB
2	01:00	480-1650	2.2GiB
3	01:30	10200-11105	1.7GiB
4	01:45	4800-5705	1.7GiB
5	02:15	11100-12005	1.7GiB
6	02:30	5700-6605	1.7GiB
7	03:00	12000-12905	1.7GiB
8	03:15	6600-7505	1.7GiB
9	03:45	16600-17505	1.7GiB
10	04:00	7500-8405	1.7GiB
11	04:30	17500-18100	1.1GiB
12	04:45	8400-9305	1.7GiB
13	05:15	3000-3905	1.7GiB
14	05:30	9300-10205	1.7GiB

15	06:00	3900-4805	1.7GiB
16	06:30	1-300	586.7MiB
17	06:45	480-1650	2.2GiB
18	07:15	10200-11105	1.7GiB
19	07:30	4800-5705	1.7GiB
20	08:00	11100-12005	1.7GiB
21	08:15	5700-6605	1.7GiB
22	08:45	12000-12905	1.7GiB
23	09:00	6600-7505	1.7GiB
24	09:30	16600-17505	1.7GiB
25	09:45	7500-8405	1.7GiB
26	10:15	17500-18100	1.1GiB
27	10:30	8400-9305	1.7GiB
28	11:00	3000-3905	1.7GiB
29	11:15	9300-10205	1.7GiB
30	11:45	3900-4805	1.7GiB
31	12:15	1-300	586.7MiB
32	12:30	480-1650	2.2GiB
33	13:00	10200-11105	1.7GiB
34	13:15	4800-5705	1.7GiB
35	13:45	11100-12005	1.7GiB
36	14:00	5700-6605	1.7GiB
37	14:30	12000-12905	1.7GiB
38	14:45	6600-7505	1.7GiB
39	15:15	16600-17505	1.7GiB
40	15:30	7500-8405	1.7GiB
41	16:00	17500-18100	1.1GiB
42	16:15	8400-9305	1.7GiB
43	16:45	3000-3905	1.7GiB
44	17:00	9300-10205	1.7GiB
45	17:30	3900-4805	1.7GiB
46	18:00	1-300	586.7MiB
47	18:15	480-1650	2.2GiB
48	18:45	10200-11105	1.7GiB
49	19:00	4800-5705	1.7GiB
50	19:30	11100-12005	1.7GiB
51	19:45	5700-6605	1.7GiB
52	20:15	12000-12905	1.7GiB
53	20:30	6600-7505	1.7GiB
54	20:45	6600-7505	1.7GiB
55	21:00	16600-17505	1.7GiB
56	21:15	7500-8405	1.7GiB
57	21:45	17500-18100	1.1GiB
58	22:00	8400-9305	1.7GiB
59	22:30	3000-3905	1.7GiB
60	22:45	9300-10205	1.7GiB
61	23:15	3900-4805	1.7GiB
62	23:45	1-300	586.7MiB

B COMMAND LINE OPTIONS

Algorithm	**Command**
bzip	bzip2 -k -v $file
gzip	gzip -6 -v < $file > $file.gz
	gzip -7 -v < $file > $file.gz
	gzip -8 -v < $file > $file.gz
	gzip -9 -v < $file > $file.gz
zopfli	/builds/zopfli/zopfli -v –i5 $file
	/builds/zopfli/zopfli -v –i15 $file
zstd	/builds/zstd/zstd -3 $file
	/builds/zstd/zstd -8 $file
	/builds/zstd/zstd -12 $file
	/builds/zstd/zstd -19 $file
	/builds/zstd/zstd –ultra -22 $file
zpaq	/builds/zpaq/zpaq a $file.zpaq $file -m1
	/builds/zpaq/zpaq a $file.zpaq $file -m3
	/builds/zpaq/zpaq a $file.zpaq $file -m5
xz	/builds/xz-5.2.9/bin/xz.sh -k -v -0 $file
	/builds/xz-5.2.9/bin/xz.sh -k -v -3 $file
	/builds/xz-5.2.9/bin/xz.sh -k -v -6 $file
xz (extreme)	/builds/xz-5.2.9/bin/xz.sh -k -v -3 -e $file
	/builds/xz-5.2.9/bin/xz.sh -k -v -6 -e $file
lz4	/builds/lz4/lz4 -1 $file
	/builds/lz4/lz4 -4 $file
	/builds/lz4/lz4 -7 $file
	/builds/lz4/lz4 -9 $file

C BENCHMARK RESULTS OF VARIOUS COMMAND LINE PARAMETERS

algorithm	size (average)	time (average)	size (min)	time (for min ratio)	size (max)	time (for max ratio)
bzip	6.47%	50.91	13.85%	80.09	1.00%	26.03
gzip 6	8.32%	43.47	16.85%	72.08	1.62%	15.02
gzip 7	8.21%	59.92	16.71%	99.10	1.59%	18.02
gzip 8	8.00%	177.90	16.45%	284.30	1.50%	40.04
gzip 9	7.92%	351.79	16.34%	531.53	1.49%	57.06
lz4 1	15.43%	3.43	29.35%	5.01	3.56%	2.00
lz4 4	11.99%	21.59	23.53%	31.03	2.58%	10.01
lz4 7	10.79%	64.50	21.60%	90.09	2.23%	25.03
lz4 9	10.34%	154.16	20.96%	218.22	2.09%	38.04
xz 0	8.19%	46.33	16.53%	80.08	1.57%	14.02
xz 3	7.83%	104.54	15.97%	219.23	1.41%	23.03
xz 3 extreme	1.54%	526.96	1.98%	502.52	1.10%	186.19
xz 6	4.12%	443.85	6.35%	523.54	1.12%	119.12
xz 6 extreme	1.10%	554.47	1.10%	281.30	1.10%	281.30
zpaq 1	9.36%	17.16	18.61%	19.02	1.79%	11.01
zpaq 3	5.63%	40.61	12.08%	46.05	0.89%	29.03
zpaq 5	5.39%	354.36	11.57%	409.42	0.84%	196.20
zstd 12	8.02%	50.05	16.40%	86.09	1.39%	13.02
zstd 19	4.18%	403.24	6.54%	426.44	1.20%	72.08
zstd 22 (ultra)	1.62%	545.26	2.06%	563.59	1.18%	253.26
zstd 3	8.99%	5.29	17.84%	9.01	1.78%	2.00
zstd 8	8.13%	24.31	16.59%	43.05	1.42%	6.01

Note that XZ seems to have very efficient results, but it did not manage to finish compressing all files in time. Because of this, its average consists only of the files that were easier to compress.