Comparative study of compression algorithms on time series data for IoT devices

FREDRIK BJÄRÅS
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Comparative study of compression algorithms on time series data for IoT devices

Fredrik Bjärås
fredrik.bjaras@live.se

Department of Electrical and Information Technology
Lund University

Supervisor: Stefan Höst (EIT) and Anders Isberg (SONY)

Examiner: Thomas Johansson

July 2, 2019

© 2019
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This Master's thesis evaluates the performance of lightweight compression algorithm aimed for IoT sensor devices. These devices are most often battery driven and produce large amounts of sensor data which is sent wirelessly. Compressing the data could be a tool in decreasing the power usage of these devices. The algorithms were evaluated based on their compression ratio, memory consumption and CPU usage. The results showed that the algorithms DRH, improved RLBE and A-LZSS performed the best according to different criteria. DRH gave the most balanced performance on all metrics, while improved RLBE is aimed towards datasets with steeper changes, and A-LZSS towards datasets with reoccurring sequences.

Popular Science Summary

With the inter-connectivity that comes with our new digital age, where smaller devices and faster network speeds make it possible to connect a wide range of devices into the so called Internet of Things (IoT), new challenges arise which have to be solved. Small wireless IoT sensor devices which contain limited hardware and run on battery power, create a challenge in providing maximum functionality while maintaining a long battery life for these devices. To reduce the power usage, minimizing the data sent from these devices is a vital part in this challenge, and compression of the data itself provides a possible solution.

Data compression is the act of reducing the size of data by changing its representation, in this case the data is values measured by the sensors of the devices. Compression methods involve a wide range of techniques, such as modifying values into smaller representations, finding patterns that repeat and replacing them with codes, and mathematical functions that modify signals, among others.

The idea behind reducing power usage by compression, stems from the fact that it may require less energy to compress and send the smaller amount of compressed data, than sending the unmodified larger data. This difference relies on how much smaller the data can be made, as well as how much energy is used for the computations. Therefore if the algorithms under-perform, they may instead require more energy than not modifying the data in the first place.

For compression algorithms to be able to operate on these devices, they have to adhere to the constraints set by the devices. Meaning they have to use a limited amount of memory, be fast and efficient, as well as compress the data sufficiently enough as to motivate the added computations. The most contradictory part of finding modern solutions to this problem, may be that combinations of classic methods reign supreme, while newer techniques mostly are too demanding. Most algorithms for this application therefore build on older methods, which are combined in new ways to form a sequential algorithm.

Of the five algorithms which were tested, it was found that they performed differently depending on the type of data the sensors collected. This shows the importance a motivated selection process, so that the proper algorithm is applied for the correct task.

Acknowledgements

This thesis was done in cooperation with Sony Mobile in Lund. Sony described the original goals of the thesis, provided datasets, equipment and valuable feedback and inspiration. Thanks to all the Sony employees that aided in the making of this thesis, with special thanks to my supervisor Anders Isberg.

Supervisor at LTH was Stefan Höst, which assisted in initial directions, feedback and thesis writing. Thank you for all the assistance.

Final thanks go out to my partner Ivi Pettersson who motivated and helped me in performing my best, all throughout the thesis work as well as my studies overall.

Table of Contents

List of Figures

Chapter 1
Introduction

Data compression is a field of information science dating back to the mid 1900s, with its introduction by Claude Shannon [1]. It revolves around the modification of data, to reform it into a reduced representation. It has always been a vital tool in ensuring that data constraints can be met, both for storage and transfer. While storage size has evolved dramatically over time [2], the increase in collection of data, especially recently with the advent of big data and machine learning, has made compression a necessity when transferring this data. Compression has always been a computationally demanding action, and is therefore often a luxury in resource scarce systems. Therefore, to motivate data compression for these systems, great optimization and effectivity is needed.

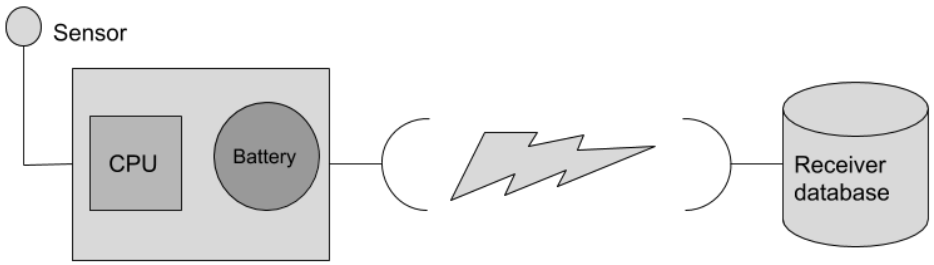


Figure 1.1: Illustration of a typical IoT sensor-device environment

IoT devices is a term for a wide range of devices and can be defined as: "the network of physical objects or things embedded with electronics, software, sensors, and connectivity to enable objects to exchange data with the manufacturer, operator and/or other connected devices." [3].

A sub category of these are small, low performance, battery powered devices (IoT devices, for this paper), and these devices are often focused on a few tasks, most often in collecting and broadcasting sensor data. These devices are meant to provide some extended service but still require little maintenance and upkeep. The sensor data is often discrete and highly regular, and may be taken in predefined

intervals, recording the data and a timestamp. Since these devices work on battery power, reducing the usage of the battery will lead to a longer lifetime and less maintenance. A large drain on the battery comes from the transmission of this data, thus a possible way to reduce battery usage is to decrease the amount of data sent from the device, namely by compressing the data on the device. Since the energy consumption of sending a byte for many IoT devices is roughly a thousand times more power consuming than a clock cycle, an effective algorithm would create a large saving in power usage of the device [4][5]. Compression is also useful for devices which are isolated, or by other reasons not send data regularly. Compression will then allow them to store more data, without adding larger amounts of storage on the devices.

1.1 Goal

The aim of this thesis was to explore the range of lossless compression algorithms and find those who can balance the resources provided on these devices. It should also explore possible implementations of these algorithms and provide a structure for balancing different performance conditions of the devices. The performance conditions in this study will be CPU usage, RAM usage and compression ratio. A secondary goal is to illustrate some relationship between the data and the types of datasets.

Research questions this thesis aims to answer:

- What is the state of the art in lossless compression algorithms for low resource devices?
- In what ways can these algorithms be implemented to optimize different performance criteria?
- How can we decide what algorithm to use, depending on the type of data?

1.2 Related work

While there is a large amount of research done in the field of data compression in general, the studies of compression on low resource devices and time series data is limited or narrow.

Gupta et al. [6], conducted a comparative study over a large range of modern compression algorithms. These were run on multiple types of files, most being some sort of text format such as plain text and source files. The study had no general restrictions on the algorithms, such as them being lightweight, and benchmarked to compute compression ratio and compression and decompression speed. The study provides a point of comparison as to what is the upper bound of compression algorithms when limited restrictions are set.

Shuai et al. [7] made a comparison between both classical and modern compression algorithms aimed at embedded systems in cars, which focuses the study

on limiting the resource usage of the algorithms. The study is unclear in if the files are compressed and compared as files or numerical values, making drawing parallels to this thesis's results complicated. The paper shows that the algorithms, Prediction By Partial Matching, Burrows-Wheeler Transform and LZ77 are the best balances of compression ratio versus the resource requirements.

Comparisons between Huffman and LZW for text, image and audio files are done in Reference [8] and [9]. These studies show inconclusive results in what algorithm is preferred for each file type.

A compression algorithm aimed at timeseries compression on IoT devices is RAKE, proposed by Campobello et al. [10]. It was initially included as an algorithm for this thesis, but was removed due to time constraints. RAKE focuses on compression sparse time series and proposes ways of making input data sparse. It shows some promise in it's simplicity and compression ratio and might perform well for specific types of datasets.

Chapter 2
Theoretical background

From previous research it becomes clear that although new compression algorithms are developed, these do not always outperform the classical algorithms in terms of balance between compression ratio and performance, as shown by Shua et al. [7]. This leads to the need for more research and studies into how these algorithms perform in specific environments, and how they can be properly optimized.

The field of compression algorithms can roughly be divided into four parts: variable length codes, statistical methods, dictionary methods, and signal methods [11].

Variable length codes are ways of representing a value as a code with a non fixed length. An example of such a method the Fibonacci code, which encodes a number into a Fibonacci representation of it, as discussed below.

Statistical methods are methods that use variable length codes which depend on the statistical nature of the input sequence. Examples of these are entropy coders, which takes the fixed length input values and encodes them as variable length codes based on their probabilities, and are discussed below.

Dictionary methods are methods which store some sort of history of the previous inputs and encodings, and encode new inputs as references to this dictionary.

Signal methods are a broad range of methods that all are used to compress signals instead of discrete values, mostly by transformations.

2.1 Definitions

To avoid any confusion in the terms used in this thesis, some ambiguities and more clearly defined in this section.

Definition 2.1 **Input value/Input**. An input value or input is an integer value $x_i \in S$ where S is the range of values for the given amount of bits.

Definition 2.2 **Time series**. A time series T is a number of input values, all

spaced at uniform discrete time intervals. All input values in a time series have the same bit length.

Definition 2.3 **Dataset**. A dataset is a collection of one or more time series.

Definition 2.4 **Character**. A character is an input value represented by one byte. The relationship between text characters and their binary representation is represented by their ASCII value [22].

Definition 2.5 **Compression Ratio**. The compression ratio CR of a set of input values is the ratio between the compressed and uncompressed size in bytes, shown by Formula 2.1.

$$CR = \frac{\text{SIZE}_{\text{uncompressed}}}{\text{SIZE}_{\text{compressed}}} \tag{2.1}$$

Where the uncompressed size is defined as: The amount of input values, multiplied with the size of each value in bits, and then divided by eight to get the size in bytes, shown by Formula 2.2.

$$\text{SIZE}_{\text{uncompressed}} = \frac{\text{input\_length} * \text{bit\_size}}{8} \tag{2.2}$$

This is done to get the actual byte size of the combined input values, and not the size of the file used as input.

2.1.1 Prediction

Predictive coding is the act of estimating the next input value with a function or process, followed by using the error between the real and predicted values as the encoded value [12]. The benefit of using predictive coding is that the entropy of the input values will decrease, since the better the prediction, the closer to zero the distribution of the error will become. One of these functions is delta encoding, where each value is predicted to the previous value, thus making the error the difference of the current value and the previous value (Equation 2.3).

$$\delta_i = x_i - x_{i-1} \tag{2.3}$$

Delta encoding therefore takes advantage of the correlations between the values, resulting in the benefit of narrowing the distribution of values. Since delta encoding encodes the differences between the values, it will make the range of values larger. For instance, if the values are one byte with the range of [0,256], the deltas must therefore allow the range of [-256,256] to allow for deltas between the max and min value.

More sophisticated predictive coding methods, often called predictive filtering, involves some linear combination of a fixed number of previous values [5]. An example of this is the double-delta coding shown by Equation 2.4, where the predicted value is a combination of the two previous values.

$$x_i = 2x_{i-1} - x_{i-2} \tag{2.4}$$

2.2 Run-length encoding

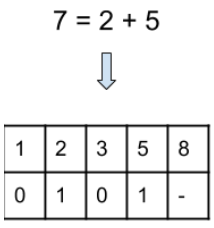
Run-length encoding [11] is a simple encoding technique that records the consecutive "runs" of a given value. If the same value occurs multiple times in sequence (a "run"), all values following the first will be represented by a count of the run length, i.e. how many times it occurs. Run-length encoding can apply to different aspect of values. Most often the value of the input or error, but it may also apply to the bits needed to represent a value or other variations of runs. Run-length encoding will most often need some sort of flag to indicate a run-length being used instead of a normal value, which will add some overhead when it is used.

2.3 Fibonacci coding

Fibonacci code is a variable length code, which transforms positive integers into Fibonacci code based on the Fibonacci numbers[11]. The Fibonacci number sequence is shown in Equation 2.5.

$$F_n = F_{n-1} + F_{n-2} \tag{2.5}$$

Each value is coded as a combination of the Fibonacci numbers that add up to the value, with a 1 appended to the end of the value to create code, as visualized in Figure 2.1.



Value	Representation	Code
1	1	11
2	2	011
3	3	0011
4	1 + 3	1011
5	5	00011
6	1 + 5	10011
7	2 + 5	01011
8	8	000011
9	1 + 8	100011

Figure 2.1: Table of Fibonacci codes and visualization of how values are encoded as them. The codes in the table are the representation of the Fibonacci numbers as shown to the left, with a 1 appended to it

2.4 Entropy coding

Entropy coding refers to techniques which encode fixed length symbols or data as unique variable length codes. The basis for entropy coding is to encode the symbols with lengths that are relative to the probabilistic distribution of the given characters. A way to calculate the entropy needed for a sequence of information is by using the Shannon entropy [1], as shown in Formula 2.6.

$$H(X) = -\sum_{i} P_i \log_2 P_i \tag{2.6}$$

$H(X)$ symbolizes the average information per symbol and P_i is the probability associated with symbol i. To calculate the entropy of a message, the probabilities of the symbols has to be calculated, followed by using the Shannon entropy (Formula 2.6) and rounding up the resulting number to a full bit, followed by multiplying this number with the length of the message.

2.4.1 Huffman coding

Huffman coding is an entropy coding technique published by David A. Huffman in 1952 [13]. The process consists of three steps. Firstly, calculate the probabilistic distribution of the characters. Secondly, create a binary tree, a so called Huffman tree. Lastly, find the un-encoded characters in the tree and encode them as the path from the root to the leaf.

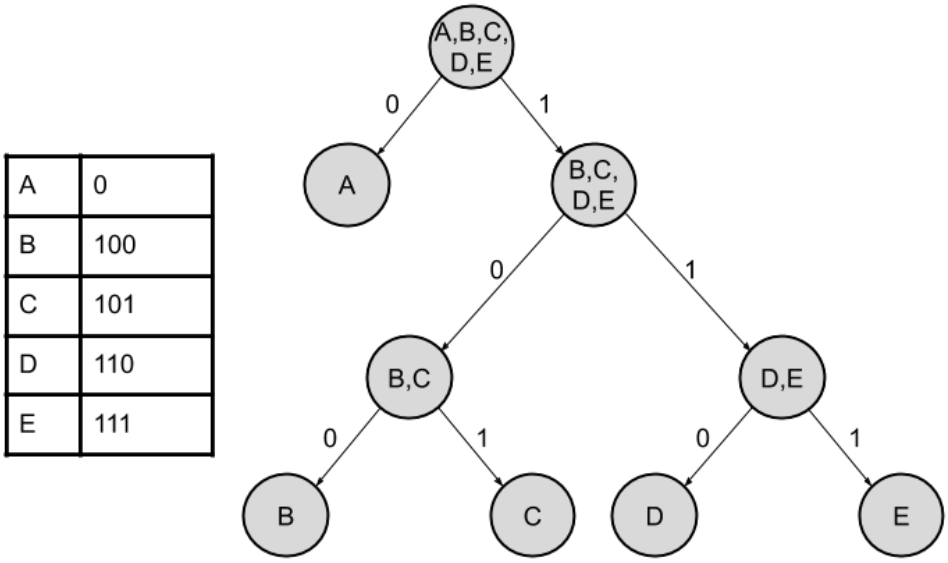
To create the tree, the first step is listing the characters according to their probabilities. The two lowest probable symbols are then put as children for a node, and the pair of them and their combined probabilities are added back to the list. This process is repeated until the list only contains one element with all characters, which is then added as the root of the tree. An example of a Huffman tree and the codes are shown in Figure 2.2.

The Huffman code of a character is found by following the path from the root to the leaf, adding a 1 if the path splits left and 0 if right.

Huffman codes is the minimal way of representing each character but for the whole message it will rarely achieve the entropy, which it can only do if the probabilities are all negative powers of 2 i.e $1/2$, $1/4$ and so on [11]. Nevertheless it is the most efficient for most cases, due to its simplicity.

2.4.2 DC Huffman Coefficients

A variation to calculating the distribution, is to use pre-defined distributions. DC Huffman coefficients (or other named variations) is one example of such a distribution, initially used in the JPEG standard [14], with a distribution centered around 0. The DC Huffman code is divided into two parts: the first being a code for a range the value belongs to, and the second is the code for the actual value in this range, illustrated in Figure 4.1. These two codes are then concatenated to



A	0
B	100
C	101
D	110
E	111

Figure 2.2: Example of a Huffman tree and the values of the encoded characters

form the Huffman code for the value. The code for a value can be computed using a lookup table, or by using a function to compute the code from the value.

Canonical Huffman code

Canonical Huffman code is a method of reformulating the Huffman tree so that the it can be rebuilt, only knowing the lengths of all the Huffman codes [11]. This is done by sorting the characters according to their Huffman code length and secondly their binary representations numerical value. The first code in the sorted list is given the code 0, with each subsequent character encoded as the following binary value, adding zeroes to the end of the codes keeping the lengths of the original codes. The tree can then be recreated by repeating this process in reverse. The advantage with canonical Huffman codes is that the Huffman tree can be sent represented by lengths of the codes, while keeping the size of each Huffman code, ergo the encoded message size the same as with normal Huffman code. The only drawback of this method is the additional computations in transforming the tree into canonical.

2.4.3 Arithmetic coding

Another entropy encoder worth mentioning is Arithmetic coding [15]. Arithmetic coding does not apply a code for each symbol, but instead applies a code for a block of symbols. This means that when the Shannon entropy for a message is for example 1.9, each character requires 2 bits of information on average, but since Arithmetic coding does not encode single symbols, this round up can be avoided

[11]. While arithmetic coding achieves a shorter code length overall, it is more demanding and therefore only used in specific cases where the difference between arithmetic and Huffman is worth the cost in computations.

2.4.4 Asymmetric numeral systems

A newer entry into entropy coders is the asymmetrical numeral systems (ANS) and its implemented variations, such as Zstandards finite state entropy (FSE). ANS and FSE can, as Arithmetic coding, achieve higher compression ratios than Huffman coding, but only in specific cases and at a slower speed [16].

2.5 Dictionary methods

Dictionary methods are compression methods that build on some sort of dictionary that stores previous occurrences or sequences so that new inputs can be referenced to already encountered values.

2.5.1 LZSS

LZSS [17] is a modification of the LZ77 algorithm [18]. It uses an adaptive dictionary, which is sometimes called a sliding window or a search buffer [19]. The dictionary is a fixed amount of previous input characters, which moves along with the current input value. Furthermore, LZSS and LZ77 use a lookahead buffer, which is a fixed number of input characters following the current input. For each string of characters from the input character and lookahead buffer, the algorithm searches the dictionary for the longest match of these input strings. If a match is found, LZSS firstly sets a flag bit to indicate a match. It then sends a pair of values with the offset to the match and the length of the match. If no match is found, the flag bit is set to indicate this, and the character is sent uncompressed. This process is illustrated in Figure 2.3.

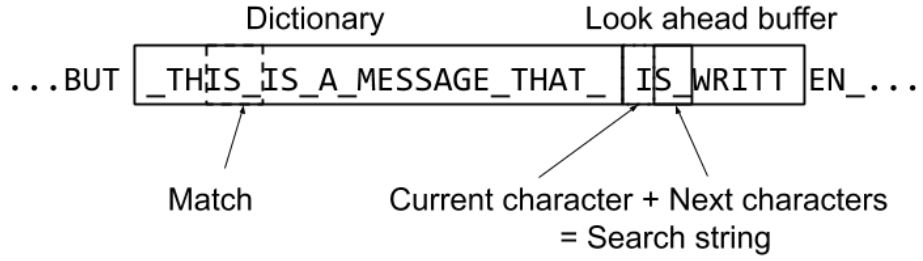


Figure 2.3: Example of a LZSS sliding window

2.5.2 LZW

LZW [20] is one of the most well known and widely used dictionary methods [11]. It is surrounded by some controversy due to its patent disputes with the GIF

formats [11]. It is a variant of the LZ78 [21] algorithm and it works by matching strings of characters to a translation table (or dictionary) as seen in Figure 2.4. The translation table is initialized with all one character strings [0,255]. For each new input character c, the following input characters are appended to c to form the string w. Characters are appended to w until a match is no longer found in the translation table. The algorithm then goes on adding the new w string to the table, and encodes the latest match as a reference to the table index. This means that for LZW, all strings will be encoded with the same sized reference. The dictionary size d is directly linked with the encoded size of the strings n, as seen in Formula 2.7.

$$2^n = d \Longleftrightarrow \log_2 d = n \tag{2.7}$$

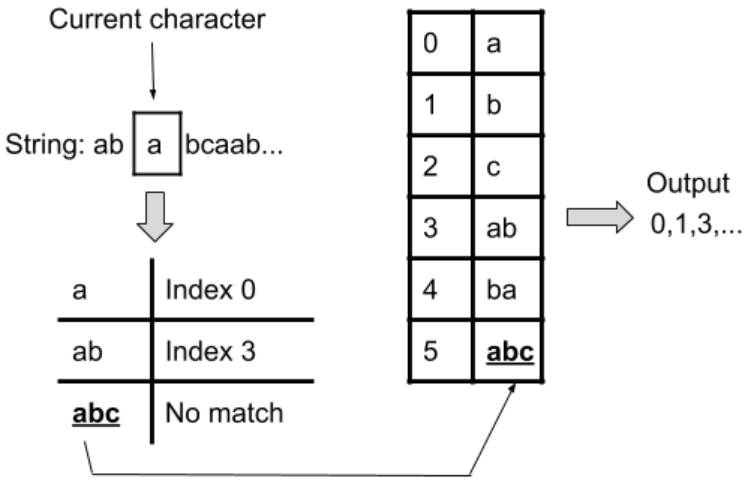


Figure 2.4: Visualization of the LZW encoding process with the alphabet a,b,c

Chapter 3
Method

Since the aim of this thesis is to evaluate compression algorithms, and their suitability for different criteria, it is important to first gain a broad, and then specific understanding of the field. This section will discuss the process of the thesis work and how it was planned and performed.

3.1 Theoretical study

The theoretical study aimed to study the field of research into the subject, both through a classical understanding of compression and through recent studies into lightweight compression algorithms. The aim of this part was to create a motivated list of candidates which could be used for further studies and benchmarking.

The background research of the field was mainly retrieved from data compression books which provided an overview, as well as discussions with the supervisor as to the common solutions used. Following this, research papers which deemed relevant or provided new algorithms were explored. The focus was on finding newer papers which aimed towards similar devices as for this thesis. These algorithms in the papers were dissected down to their components, which were examined in more detail. After analyzing and understanding the different algorithms, a few were chosen according to how well the applied to the constraints of the subject. It was decided to show a range of different methods of compression, so too similar algorithms were discarded, as well as algorithms which were specific to one or few types of situations or data.

3.2 Implementation

The implementation phase aimed to implement the candidates using similar conditions. The implementations were coded in C, and avoided all possible overhead and extension usage. This to ensure that only the integral parts were benchmarked, and to avoid conflicts when applying them to an embedded system. The implementations were made according to the specifications in the papers of the corresponding algorithms, and therefore took some necessary assumptions and liberties when interpreting these specifications. The implementations were optimized

as equally as possible, and different variations of the implementations were experimented with. The final algorithms aimed to approach a middle ground of the performance metrics measured.

3.3 Evaluation

The last part of this study was to evaluate the results. The evaluation focused on both the individual algorithms and their performance metrics CR, memory and CPU on the different datasets, as well as a comparison between the algorithms. The results were gathered from a benchmarking framework which could run a specific algorithm on a dataset in a fixed configuration. Each algorithm was run through the same datasets and configurations to provide the same metrics. These metrics were then summarised into the results.

Chapter 4
Candidates

This section describes the chosen candidate algorithms. The reasoning for choosing the algorithms may differ, but the overall goal was to both get a diverse group of algorithms, and most importantly to choose algorithms which are applicable to the low resource requirements of the IoT devices the thesis aims to investigate. The applicability of the algorithms was based on the claims stated in the papers specifying the algorithms.

4.1 DRH

DRH is an abbreviation of delta, run-length and Huffman, and was proposed by Mogahed and Yakunin in 2018 [23]. They found the range of suitable lightweight compression algorithms lacking and therefore looked at what components could give an acceptable compression ratio with a low resource demand.

As the name suggests, DRH puts the input values through delta encoding, run-length encoding and Huffman coding. DRH only handles integer values, and other data types are quantized into integers. The integers are then delta encoded and successive values are checked for run-length with a possible counter added for this. Lastly, the values and counters are Huffman encoded and concatenated to form an output sequence.

The Huffman tree used by DRH is a pre-defined Huffman tree to be stored on the device. Mogahed proposes two Huffman trees, one for values with a narrow scope, and one being a DC Huffman coefficients lookup table as seen in Figure 4.1.

4.2 Sprintz

Sprintz was presented by Blalock et al. in 2018 [5], and similarly to DRH, it combines multiple simple compression techniques together in a sequence. Sprintz starts with a predictor, either delta encoding or Sprintz's own FIRE forecasting. FIRE predicts the errors as a weighted version of the previous error plus a noise term:

$$\delta_i = \alpha\delta_{i-1} + \epsilon_i$$

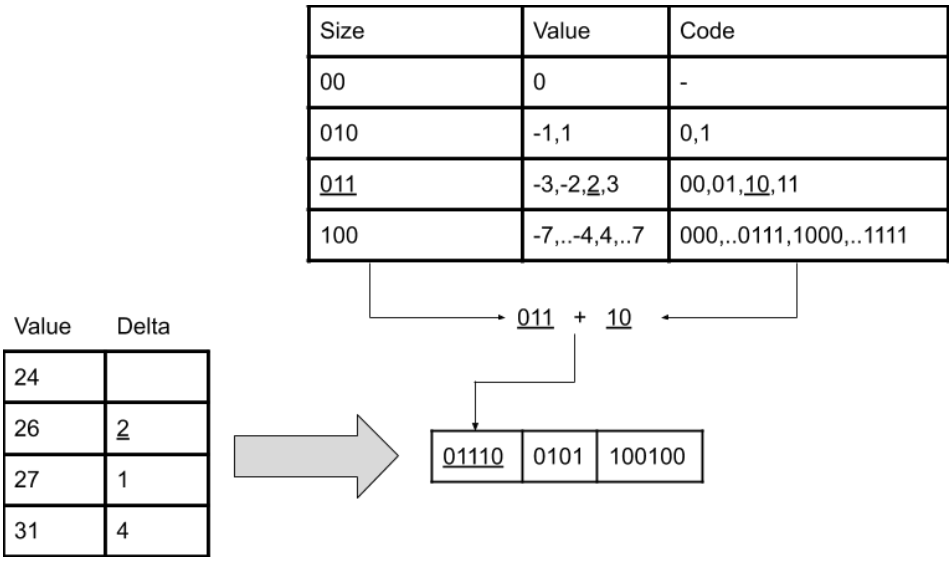


Figure 4.1: Visualization of the DRH process. Deltas encoded using the DC Huffman coefficients and concatenated into a bit stream

The weight is trained online, meaning it is updated locally on the device after each block.

After prediction, the values are zigzag encoded, meaning the range of [-a,b] is converted to the unsigned range [0,a+b] using:

$$x_i = (e_i \gg (\text{bitlength} - 1)) \oplus (e_i \ll 1)$$

After prediction and zigzag encoding, the values are checked for the maximum number of bits needed to represent all the values. A header with the bit-length of the values is created, and all the values are concatenated together, each with the size of the bit-length (Figure 4.2). The Sprintz process is done for a block size of 8 input values, meaning that the for each 8 values, there will be a header and the processed values (Figure 4.2).

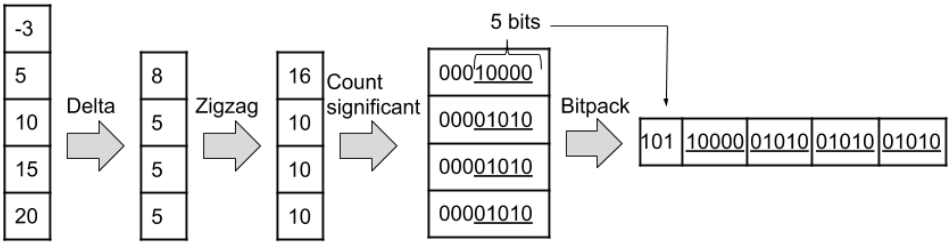


Figure 4.2: Illustration of the Sprintz process

The final part to Sprintz is a entropy encoder. Blalock et al. suggests using Huffman coding, which is calculated online, meaning on the device. Huffman is calculated using each byte of the bit-packed sequence as a character.

The different components of Sprintz are interchangeable in some configurations. Delta or FIRE can be selected according to performance requirements and Huffman can be used if higher compression ratio at a slower speed is desired.

4.3 RLBE

RLBE, or Run-length binary encoding, is a sequential algorithm, such as DRH and Sprintz, which was proposed by Spiegel et al. [24]. It predicts the values using delta encoding followed by a a run-length encoding on the values. The run-length encoding used by RLBE is not the same as for DRH, but instead compares at the number of bits needed to represent the value. If following values all have the same number of bits, the values are counted until the run-length is over. The counter is encoded as a Fibonacci code.

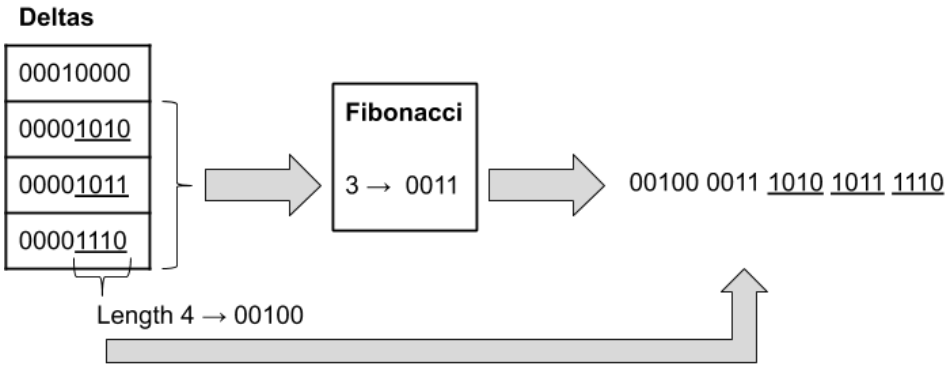


Figure 4.3: Illustration of the steps in the RLBE algorithm

The final step is to put together the components: a header describing the bit-length of the values, the Fibonacci code describing the number of values following, and the values themselves using only the needed amount of bits (Figure 4.3).

4.4 A-LZSS

A-LZSS, or Accelerometer LZSS, is a dictionary method built on the LZSS algorithm [17]. A-LZSS was described by Pope et al. [25], and works by using LZSS to look for matches. If a match is found, but the length of the matched sequence is below a certain threshold, the value is instead encoded by an offline Huffman code. Since LZSS is built to handle characters as input, the values are first delta

encoded, and then split into bytes.

The Huffman tree is calculated offline, using data from the sensor the algorithm shall operate on. The Huffman tree is converted into a lookup table to be used by the algorithm. The lookup table size will be proportional to the size of the values to be encoded, thus for a lookup tree of size 256, each character is taken as input. A larger lookup table will result in better compression, since multiple characters can be encoded with a single code, however it also requires more memory to be stored.

4.5 D-LZW

D-LZW, or Differential LZW, is a dictionary method proposed by Le and Vo in 2018 [26]. The main principle is that LZW relies on building a dictionary based on repetitions. To make most use of this, the scope of the input values is lowered by delta encoding. D-LZW does not specify the size of the dictionary, or the options for when to clear it, but it does specify that a fixed size dictionary is to be used. It also specifies that strings may have no restrictions on their length.

4.6 Modifications to the candidates

While most algorithms are implemented as close as the specifications as possible, some modifications and improvements were still made, and will be documented here.

For DRH, the DC Huffman coefficients lookup table was used. To save memory this tree was not saved in the device, but instead a function to calculate the code was used. Due to the use of run-length encoding, some type of flag was needed to distinguish the run-length counters from the normal values if the whole output is sent as a block. This was experimented with in a number of ways, but in the end run-length encoding was removed due to the datasets containing too few repeats of deltas to motivate the use.

Sprintz is implemented in four variants: with or without Huffman, and with delta encoding or FIRE. Only one dimension of values is used, instead of the multi-variable method as suggested by the paper. Additionally a offline Huffman code was experimented with, but not presented in the results.

RLBE encoding is implemented using zigzag encoding for the deltas, such as in Sprintz. Additionally, the run-length is bound to avoid large Fibonacci codes, since these are stored in an array in the device. An improvement to RLBE is also implemented, which allows for an interval of values to be represented with a higher bit-length. This is done to avoid the overhead for that come with each short run-length by combining them. This improved algorithm is denoted as "improved RLBE" in the results.

A-LZSS is implemented using the deltas as the values instead of the actual input values. The offline Huffman code is the same as for DRH. Experimentation with an offline Huffman code based on the data distribution was performed, but not used in the final implementation. The implementation also does not divide up values into bytes, since DC coefficients allows for any sized value to be encoded.

D-LZW does not specify the actual implementation of the dictionary more than as an array of the stored strings. This was implemented as an array of a (previous index, char) pair. The character is the last character of the string, and the previous index variable points to the array index of the previous character. This makes it possible to avoid storing the whole string in the array, and was inspired by Juha Nieminen [27].

Chapter 5
Results

This section will present the results of the benchmarking, define the environment, and specify how the datasets were used.

5.1 Implementation specifications

The specifications of the system running the implementations: Ubuntu 16.04 LTS, with a Intel® Core™ i7-4600U CPU @ 2.10GHz x 4 CPU, and 15GB in RAM memory.

The code was compiled with gcc 5.4.0. using the optimization flag -Os. Heap memory usage of the algorithms was measured from calculating the current allocations made by the algorithm and saving the maximum. Stack memory was measured by the -fstack-usage option in gcc and calculating the maximum stack usage from the deepest functions. CPU estimates were calculated using C standards <time.h> library with the functions clock() and the constant CLOCKS_PER_SEC.

The implementation was aimed at an IoT device using an ARM Cortex m4 processor, with available RAM of about 28kB.

5.2 Datasets

Multiple datasets are used for benchmarking the algorithms:

- Pig positioning dataset
  Accelerometer data, taken from sensors placed on pigs. Data is divided into x,y and z values, each 10bits long. Distribution for the deltas of the values are shown in Figure 5.3. The dataset contains multiple files, each file corresponding to a separate device.

- Blood glucose dataset
  Blood glucose data dataset. Values are floats, quantizised to 16bit integers. Distribution for the deltas of the values are shown in Figure 5.2. This dataset contains 849 values in one time series.

- Tracking dataset
  Accelerometer data from GPS tracking project. Values are floats, quantizised to 16bit integers. Distribution for the deltas of the values are shown in Figure 5.1. The dataset contains multiple independent time series.

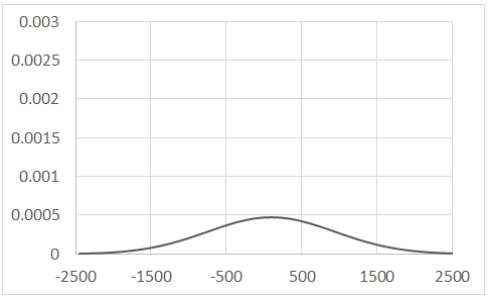


Figure 5.1: Distribution of deltas from the tracking dataset

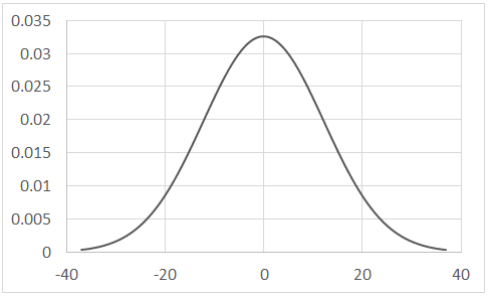


Figure 5.2: Distribution of deltas from the Blood glucose dataset

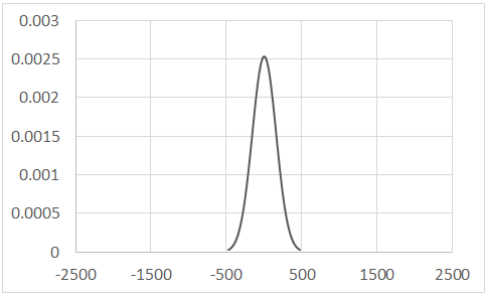


Figure 5.3: Distribution of deltas from the pig dataset

5.3 Measurements

The benchmarking tool runs the algorithm with a dataset, a specified amount of input values, iterations and the bit length of the values. This returns the average compression ratio, the average amount of heap memory used specifically by the algorithm, and the average time elapsed for the compression.

The compression ratios for all algorithms on the pig dataset are shown in Figure 5.4. The pig data is run for 1000 iterations, resulting in a million input values for the block size of 1000 values. Each iteration uses a unique time series so to avoid large gaps between different series. The compression ratio of D-LZW is also shown in Figure 5.5 The division is due to the fact that D-LZW requires a dictionary size, and is not dependent on the input length, but instead the dictionary improves over time. D-LZW in Figure 5.4 therefore shows the average compression ratios for the dictionary sizes in Figure 5.5 for the whole input of a million values and not the individual iterations. Zlib is also run on the pig dataset, shown in Figure 5.4, to provide a baseline comparison. It was run using Mark Adlers implementation [28] with the configuration of windowBits=9 and memLevel=1 to provide a memory sparse configuration.

Figure 5.6 shows the compression ratio for the blood glucose dataset. D-LZW is excluded due to the small input data.

The compression ratios for the tracking dataset is shown in Figure 5.7. The time series for this dataset are of different lengths, thus each full time series is compressed independently.

The total memory usage for the algorithms when using the pig dataset with 1000 input values are shown in Figure 5.8. D-LZW uses a dictionary of 7172 bytes for this measurement.

The CPU usage, measured in the time each algorithm runs for, is shown in Figure 5.9. This average is based on 1000 of iterations of the pig dataset with 1000 input values for each iteration. Measuring begins when all preprocessing is done, such as reading files into lists, and the compression algorithm starts. It ends when all the values have been compressed and allocations made by the algorithm have been freed.

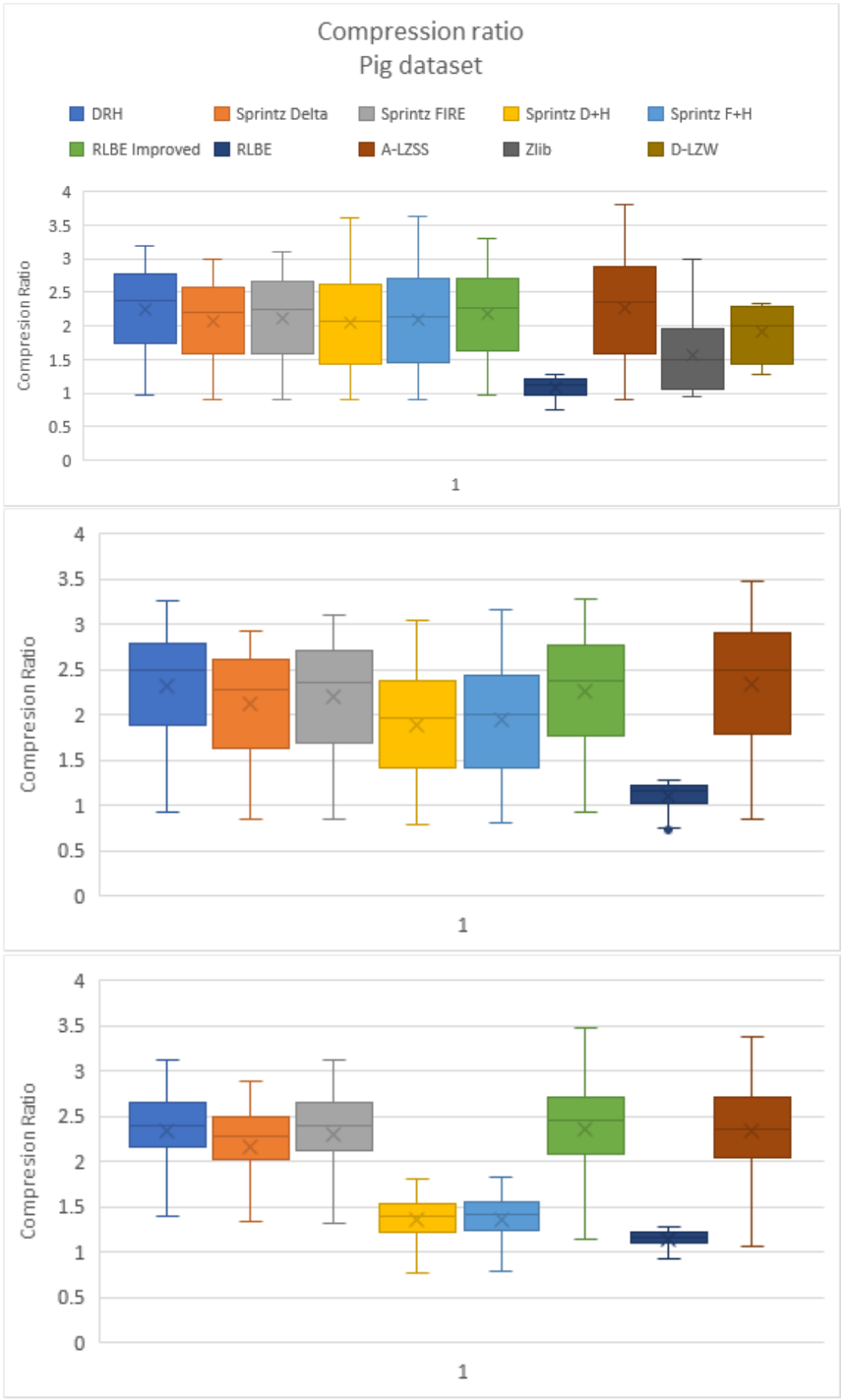


Figure 5.4: Compression ratios for algorithms run on pig dataset

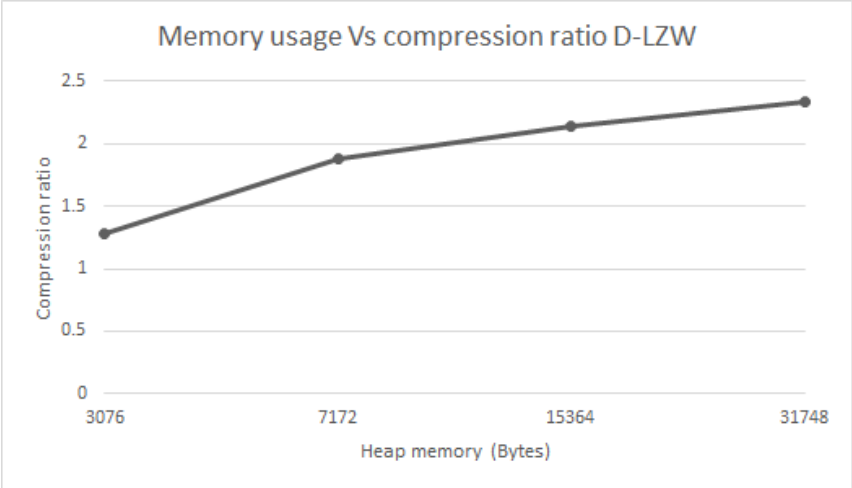


Figure 5.5: Compression ratio for D-LZW run on the pig dataset, compared to the dictionary size

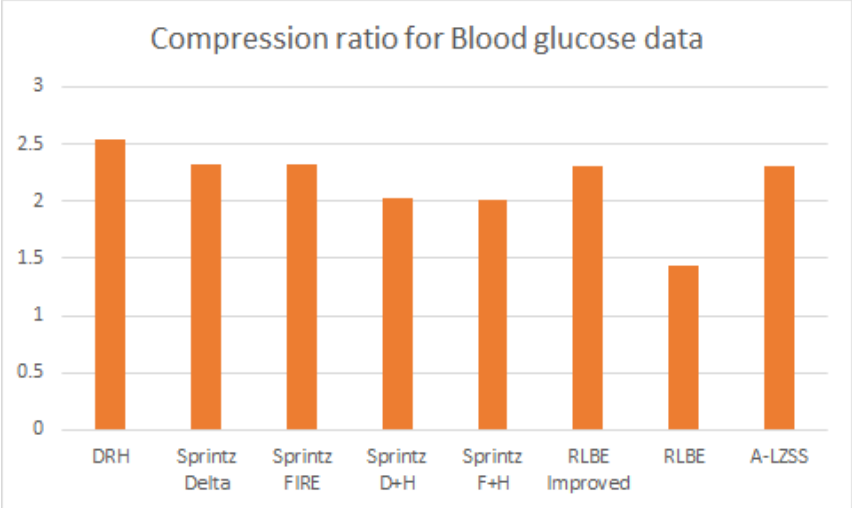


Figure 5.6: Compression ratios for algorithms run on blood glucose dataset

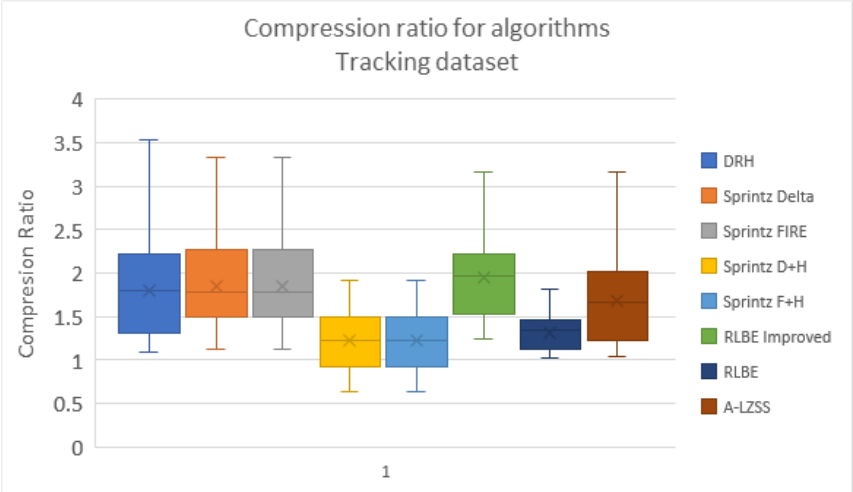


Figure 5.7: Compression ratio for algorithms run on tracking dataset

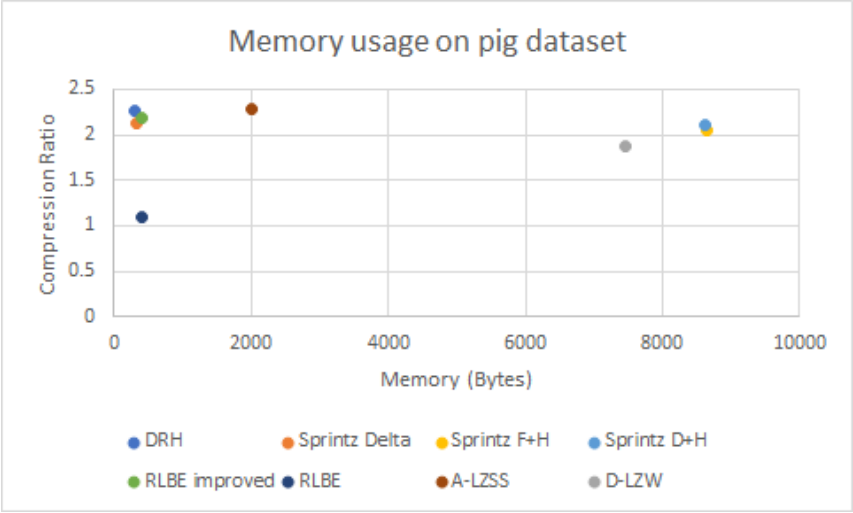


Figure 5.8: Heap and stack memory usage compared to compression ratio for pig dataset with 1000 input values

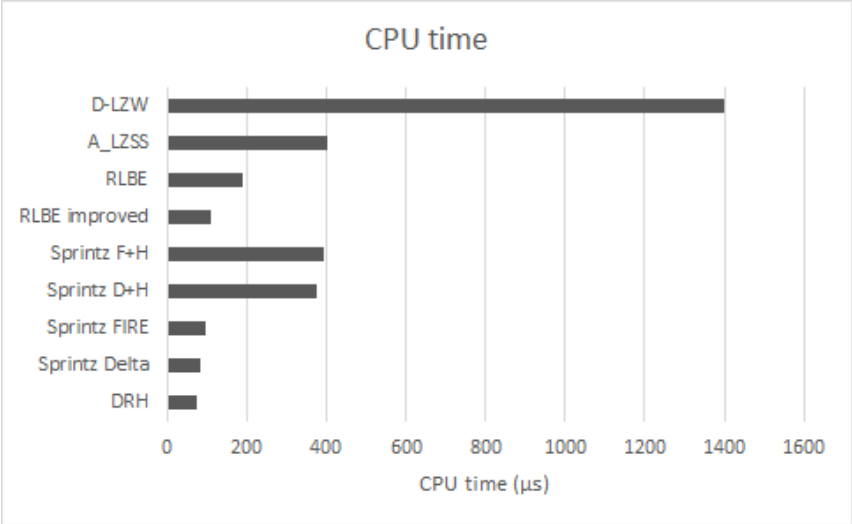


Figure 5.9: Time elapsed from starting compression to completions. Excludes overhead such as writing and reading from files. Time is calculated using the elapsed clock ticks divided by the ticks per second

Chapter 6
Discussion

This part aims to answer the research questions stated in section 1.1. The important subjects to discuss when answering these questions are: what algorithms perform the best on what datasets, why the algorithms perform better/worse in general and for specific datasets, and what conclusions can be drawn from this.

6.1 General comparison

Since there are three different variables measured and investigated (CR, memory, CPU), we firstly have to look at what these affect.

Memory usage can be seen as both the least and most important variable to look at. Due to the fixed amount of memory available on the devices, a bound for what algorithms can operate on the device is created. Figure 5.8 shows that all the algorithms require less than the 28kB available on the devices. Both Sprintz implementations using online Huffman encoding are outliers in this case, but they still are well within the margin of the requirements. As D-LZW allocates a predetermined amount of memory, it can be forced to comply with the memory requirements. This comes at the cost of the compression ratio, which can be seen in Figure 5.5. Even though all algorithms can comply with given memory requirements it does not mean that it should be ignored. Some devices may have less memory available than the value on the target device. Smaller usage of memory may also allow for other processes to run on the devices, and therefore make the algorithm more viable over time, with continuous changes and additions to the device.

Regarding compression ratios, it appears that algorithms have an overall compression ratio higher than 1 for all the datasets, and therefore produce an output smaller than the input. Nevertheless, there are some substantial differences between the compression ratios. DRH, improved RLBE and A-LZSS are the highest compressing algorithms in general, with DRH topping the list overall. The standard RLBE has the lowest compression ratio with a wide margin. Zlib provided a worse compression ratio than all the algorithms except standard RLBE. This might be due to the fact that is was used "out of the box", while all other algorithms were specialized and modified to fit the datasets. Other configurations

may also provide better results, but it still acts as a baseline for what a regular compression algorithm can perform with these constraints.

As for CPU performance, D-LZW is roughly 8 times slower than the bulk of the algorithms lying in the vicinity of 1400 microseconds. Both Sprintz algorithms using the online Huffman encoding are also substantially slower than the others.

6.2 DRH

DRH was the overall best performing algorithm of the range. It provided some of the highest compression on all the datasets, while requiring a low amount of memory and achieving a fast speed.

DRH bases its compression on first centering the distribution of the values around zero using delta encoding, followed by using DC coefficients to decrease the sizes of the values. Normal Huffman encoding is built on encoding the characters according to their distribution. The chosen DC Huffman coefficients codes, are based on a distribution centered around 0, with increasing code lengths when the value strays from the center. Using DC coefficients is a trade-off between memory and compression ratio. Storing a pre-calculated distribution on the device will take up some memory. For input values in the range of one byte, 2^8 (256) codes need to be stored. Increasing the range of the input values to two bytes will then require 2^{16} (65536) codes, meaning that it becomes a matter of either storing large Huffman trees, or dividing each value into smaller components (bytes). Dividing for example a 16bit 3, into 0 and 3. This may provide a worse compression ratio, due to each part needing to be encoded separately. Instead, by using DC coefficients, which can be computed using a fixed function, it means that a distribution not has to be stored. DC coefficients also allows integer variables with any bit-size to have the same code when their value is the same.

Both memory and CPU usage is low for DRH. It does not require any data structures to be allocated, and simply keeps some limited variables used in the calculations on the stack. The calculations are mainly limited to the delta encoding and the calculation of the Huffman codes for the values, which does not add up to any substantial amount of CPU instructions.

It is hard to compare the results to the findings of Mogahed [23], due to the fact that the datasets used in that study all have a much narrower standard deviation of the deltas. For instance, one dataset contains 32 bit values, and almost all the deltas lie in the range of [-3,3], Huffman encoding the values as maximally 5 bits each. Compare this to to pig dataset in Figure 5.3, with 10 bit values and a much larger spread for the deltas.

6.3 Sprintz

Sprintz is a slightly more sophisticated compression algorithm than DRH, and can be varied in some ways that make it perform differently. Firstly, the variations without Huffman encoding both perform above average for most of the datasets, while having the highest CR for the tracking dataset as shown in Figure 5.7. The variants using Huffman encoding perform below the average, as well as below the variants without Huffman, for all but the pig dataset with 1000 input values. FIRE prediction provides a minor improvement compared to delta encoding for the pig dataset, both with Huffman and without.

Sprintz is built on removing the leading zeroes of the deltas and encoding each internal block with the maximum bit size of these values. It also adds a small header to each block. This makes it so that Sprintz is heavily dependent on the correlation between values, as stated by Blalock et al. [5]. This dependency stems in the fact that each internal block of eight values is encoded as the maximum size of the values. Consequently, if the deltas are [1,2,1,2,1,65,2,0] for a block, all values will be coded as 7 bits long, instead of the 2 bits if 65 would have been for example 1 instead. This makes the Sprintz block 7*8 + 3 = 59 bits. Comparing this to the DC Huffman coefficients, which is 41 bits, shows that large changes in deltas have a negative effect on the compression ratio. Even though Sprintz will under-perform when the deltas go towards the extreme, it still handles the middle range, such as the tracking dataset in Figure 5.1, better than the other algorithms, which can be seen in Figure 5.7. This is because the correlations of the inputs do not have to be centered around a common point, as for DC coefficients, and instead relate to the similarities in lengths for nearby deltas.

When the blocks have been concatenated, they form a byte sequence without any strong correlations between the blocks, due to the differences in length. This creates a more uniform distribution which results in little compression due to Shannon entropy. It still has the possibility to provide some additional compression, and will never increase the size [11]. Looking at Figure 5.4, this would appear to not be the case, since the variants with Huffman have a lower average. The explanation behind this result is the overhead that comes with using an online Huffman. The information on the Huffman codes has to be expressed in some way, which is implemented using canonical Huffman codes, sending the lengths of the codes along with the compressed data. This is clearly shown for the smaller input lengths in Figure 5.4. Ways of mitigating this overhead could be to increase the input size, calculate the distribution for multiple inputs and save it, or use adaptive Huffman coding.

Sprintz with no Huffman encoding is fast and memory efficient, both using delta and FIRE encoding. Both of these variants require few memory allocations, and only limited logical computations such as additions, multiplications and bit shifting for the main loop, which are all 1 cycle instructions [4]. The Huffman variant is originally the same, but applies a Huffman encoding that reforms the byte sequence. This adds the complexity of both iterating through the message to cal-

culate the distribution, creating the Huffman tree, and converting it to canonical Huffman code. The message has to be iterated through again, converting the bytes to Huffman codes. For memory, this requires the whole input sequence to be stored in memory, and the Huffman tree to be stored. There is probably room for improvements in the implementation specifics for both memory and CPU, but it will always require substantially more resources than the non-Huffman variants.

In the study by Blalock et al., Sprintz is measured using the UCR dataset, for 8 and 16 bit integers [5]. The dataset contains various types of time series with different lengths, which are each run and plotted as a distribution of the CR's. These results are similar, but somewhat lower than the results presented in this thesis. Moreover the details of the Huffman encoding are not clearly specified, making comparisons between these results complicated.

6.4 RLBE

The RLBE algorithm is the worst performing algorithm when looking at compression ratio. For memory and CPU it ranks among the best, only requiring a small amount of heap memory to store the values for each run-length. The improved variant increases the compression ratio, ranking it among the top for all datasets, and best for the tracking dataset. It is also is faster than the normal variant and uses the same amount of memory.

RLBE encoding is based on the successive bit-lengths of following deltas. Successive deltas of the same bit-lengths can be concatenated together, adding a header with the length of the run and the size of the deltas. The compression ratio is therefore ultimately dependent on sequences of deltas that share a most significant bit. Higher value deltas have a larger probability of having the same most significant bit, due to the increased range of values that it represents, making the likelihood of run-lengths higher for larger deltas. To illustrate this, take the value 85 which has its most significant bit on the seventh position, making it a length of seven. This gives the numbers [64,127] the same length. To generalize this relationship: for a binary value with the length of n, there are 2^{n-1} values with the same length. Hence, for distributions around zero, such as Figure 5.3, the length of the values will constantly change, leading to short run-lengths and more overhead. This is the reason why it performs better for the tracking dataset in Figure 5.7. The improved RLBE algorithm removes some of the downsides that comes with RLBE, namely that when values often change length back and forth, no longer run-lengths are created. This is removed by allowing shorter lengths to be encoded as longer, and therefore avoiding the lengthy headers that come with each new run-length. A possible downside with this solution is the possibility of increased bit-lengths for run-lengths in certain cases. An example of this is shown in Figure 6.1: the length for the original is 37 bits, which is smaller than the improved versions of 39 bits, and in addition, with longer run-lengths this difference will only increase. However, improved RLBE still takes advantage of what makes RLBE compress, and should in general only provide a more versatile algorithm

Bit length	Run-length Fibonacci code	Value 1	Value 2	Value 3	Value 4	Value 5	Sum (bits)
2 = 00010	1 = 11	3 = 11			= 9		9
4 = 00100	5 = 00011	9 = 1001	3 = 0011	3 = 0011	3 = 0011	3 = 0011	30

Bit length	Run-length Fibonacci code	Value 1	Value 2	Value 3	Value 4	Value 5	Sum (bits)
2 = 00010	1 = 11	3 = 11					9
4 = 00100	1 = 11	9 = 1001					11
2 = 00010	4 = 1011	3 = 11	3 = 11	3 = 11	3 = 11		17

Figure 6.1: Example showing the sequence 3,9,3,3,3,3 being encoded using improved RLBE (top), and standard RLBE (bottom)

with the same main strength as the original

The heap memory usage of RLBE comes from the storing of the value in a run-length until it is over. Beyond this, the Fibonacci codes for a fixed number of lengths are also stored to avoid re-computations. The CPU usage is dependent on iterating through the input values and computing the delta and bit-length. Additionally, some operations are done for each run length, such as packing together the run-length and header. This is why Figure 5.9 shows a difference between the regular and improved RLBE, since the improved will have fewer run-lengths.

Spiegel et al. [24] do not give any details about the dataset used, but due to an example in the paper with deltas of around 30, and the results from their study, it can be assumed that the dataset contains a distribution with much higher deltas than seen for the datasets in this thesis.

6.5 A-LZSS

A-LZSS performs similarly to DRH and RLBE in compression ratio for most datasets, with the exception of ranking lower for the tracking dataset as shown in Figure 5.7, while using slightly more memory and CPU.

A-LZSS builds its compression on the correlations between the input values. These correlations are both exploited by searching for previous occurrences in the input sequence, as well as by similar values resulting in deltas close to 0 used by the Huffman encoding. The reasoning for using deltas instead of the input values in the benchmarking was both to compact the distribution of values, but also to avoid the lengthy Huffman codes that larger values would result in. This dependency on similar values is shown in the results, with the tracking dataset and its wider distribution illustrated in Figure 5.1, resulting in a worse compression ratio

compared to the other algorithms. The most important aspect of A-LZSS is the chosen sizes of the dictionary and lookahead buffer. Through testing, the best compression ratio was gained from using a dictionary, lookahead buffer and minimum match of 7, 3 and 3 for the pig dataset. A larger dictionary will find more matches, but also make each encoded match longer. A longer lookahead buffer can find longer matches, but will also make the encoded matches longer. If the encoded matches are longer, the minimum match length also has to increase, so to avoid sequences that would be smaller using separate Huffman codes for the values. Furthermore, the minimum match can not be to long, since A-LZSS without any matches is basically DRH with one bit extra before each Huffman code. These parameters have to be customized for each type of dataset to accommodate for the differences in how often and how long repeating sequences are, and to get optimal compression. From the results, it is clear that not enough matches are found, with A-LZSS achieving a worse average compression ratio than DRH for all the datasets. A-LZSS could theoretically outperform DRH on datasets with more smooth and continuous changes than the ones tested, thus further studies investigating this are needed.

The memory consumption and speed of A-LZSS is directly dependent on the size of the buffers used. Increasing the buffers sizes will result in more previous values being stored and longer searches for the best match. Other than the searches in the dictionary, limited computations are performed, meaning that optimizing the search is the most vital part.

Pope et al. [25] suggest the optimal point is when using both a 4bit dictionary and lookahead buffer. These values are found when taking both compression ratio and the search times are taken into account. Pope et al. tested A-LZSS on a dataset with 8bit accelerometer data, achieving a compression ratio of 3.3. This is higher than the compression ratios achieved in this benchmarking, but still reasonable if the dataset is more suitable for A-LZSS.

6.6 D-LZW

D-LZW is compared in a slightly different way than the others. Mainly since it improves when more iterations are run and the dictionary fills up, but also since it is implemented to not run the input data in blocks, but as a stream of values. When ranked against the other algorithms in Figure 5.4, it performs among the worst. Moreover, it is the slowest algorithm by a factor of 3 to the closest one.

D-LZW builds its compression on firstly narrowing the distribution of values by delta encoding the input values, and then encoding the similarities in the data using previous occurrences of sequences. The aspect then becomes how long the sequences of data are, and how many of these can be stored. It should thereby perform well on datasets with a distribution with high frequencies close to zero, and long time series which repeat the same patterns, such as temperature data. None of the datasets in this study apply to these criteria, both due to size, and

due to the high variations in the pig dataset. Therefore, the results for D-LZW are inconclusive and further benchmarking is needed.

The relationship between the memory and the CPU usage in D-LZW is a significant factor in its performance and implementation. For each value, multiple searches are done towards the dictionary. For each new character added to the string, a new search is performed. This puts emphasis on the data structure of the dictionary in how fast a search can be performed. The solution in this implementation is focused on lowering the memory consumption of the dictionary. The dictionary thereby does not store the strings, but instead stores pairs of the character and index of the previous character. Each search is then limited to iterating through the dictionary in the range of the index of the last added pair and the index of character before the currently searched one. If the dictionary is large and recurrences are infrequent, this range will be large. Focusing more on speed, other data structures might be more suitable, but might require more memory. Another aspect of D-LZW is it is high vulnerability to data loss, due to the fact that the dictionary builds up over time and is rebuilt by repeating the algorithm in the decoding stage from the initial characters and the indexes. Meaning, if data is lost, the dictionary will be different from the encoding side, resulting in erroneous decoding.

Le and Vo [26] present much higher compression ratios than the results presented in this thesis. They use PPG and ECG signals as datasets, which have highly regular frequencies as shown in their paper. These frequently reoccurring values could give D-LZW the ideal prerequisites for high compression ratios, which might be the reason for the higher results. They also do not specify the dictionary size, and if larger dictionaries are in fact used, compression ratio could be higher. From previous research [9], it is shown that LZW has high variations in its compression. Based on this, LZW should be applied carefully to datasets, but has potential for high performance in specific circumstances.

6.7 Further research

This paper provides some insight into the choices of algorithms when aiming to compress time series data taken from IoT sensor devices. The area has a lot of need for further research, both into these algorithms themselves, and to other algorithms or approaches.

One approach would be to evaluate more advanced prediction methods. A possible tool is machine learning, to train a predictor in predicting the values more accurately. This approach could be applicable to almost all of the algorithms in this paper.

RLBE saw much potential in this paper, and increasing the range of datasets would give more information into how it can perform. The improvements to RLBE made in this thesis should also be further explored, since it has great potential if

examined more closely.

Other future work could include examining the energy consumption of the algorithms on embedded devices, compared to the metrics explored in this paper.

Chapter 7
Conclusion

When examining the combination of compression ratio, memory usage and speed, DRH, improved RLBE and A-LZSS are the overall best performing algorithms for the datasets used in this study. DRH and RLBE are both fast, and limited in memory usage, while also providing high compression ratios for all the datasets. A-LZSS has slightly higher demands in these regards, but its consistent compression ratio compared to the other makes it highly desirable as well. Sprintz without Huffman performs slightly below DRH and improved RLBE, making it sub par to these for the applied datasets. The Sprintz variants with Huffman have show a varied compression ratio, but could be suitable for larger input lengths if the higher memory consumption can be accepted. D-LZW was the lowest performing algorithm of the benchmarked. It consumes more memory than the others, and takes substantially more time.

More studies on these algorithm should be conducted, both with additional and larger datasets, and on variation on these implementations.

7.1 Algorithm recommendations

For the general case and for datasets with a narrow distribution of deltas around zero, DRH is the clear choice. This is based on the simplicity of the algorithm, the low resource requirements needed, combined with the high compression ratio for all datasets.

For datasets with steeper changes and a wider distribution of deltas, the improved RLBE is recommended. It is almost as light on resources as DRH, but outperforms it for these types of datasets. It also has high potential for specialized use on specific types of datasets, with possible room for improvements to the algorithm itself.

For datasets with repeating sequences, A-LZSS may be the best choice. It is flexible in that the parameters can be modified to suite the distance between the repeating patterns

References

[1] Shannon, C. "A mathematical theory of communication" (1948) The Bell System Technical Journal, Bell System Technical Journal, The, Bell Syst. Tech. J, (3), p. 379. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[2] "Data storage – then and now", Viewed 25/05-19, https://www.computerworld.com/article/2473980/data-storage-solutions-143723-storage-now-and-then.html

[3] IETF "The internet of Things", Viewed 24/6-19, https://www.ietf.org/topics/iot/

[4] ARM Cortex-M4 Processor Technical Reference Manual Revision r0p1, Viewed 24/5-19 `https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf?_ga=2.101962692.433764804.1558686834-1085243297.1558686834`

[5] Blalock, D., Madden, S. and Guttag, J. (2018) "Sprintz: Time Series Compression for the Internet of Things". doi: 10.1145/3264903.

[6] Gupta, A., Bansal, A. and Khanduja, V. "Modern lossless compression techniques: Review, comparison and analysis" (2017) 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Electrical, Computer and Communication Technologies (ICECCT), 2017 Second International Conference on, p. 1. doi: 10.1109/ICECCT.2017.8117850.

[7] Shuai, C., Li, S. and Liu, H. "Comparison of compression algorithms on vehicle communications system" (2015) 2015 IEEE Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), 2015 IEEE, p. 91. doi: 10.1109/IAEAC.2015.7428525.

[8] Al-laham, M., El Emary, I. "Comparative Study between Various Algorithms of Data Compression Techniques" (2007) IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.4, April 2007

[9] Bedruz, R A., Quiros, A R. "Comparison of Huffman Algorithm and Lempel-Ziv Algorithm for audio, image and text compression" (2015) 2015 International Conference on Humanoid, Nanotechnology, Information Technology,Communication and Control, Environment and Management (HNICEM),

Humanoid, Nanotechnology, Information Technology,Communication and Control, Environment and Management (HNICEM), 2015 International Conference on, p. 1. doi: 10.1109/HNICEM.2015.7393210.

[10] Campobello, G., et al. "RAKE: A simple and efficient lossless compression algorithm for the Internet of Things" (2017) 2017 25th European Signal Processing Conference (EUSIPCO), Signal Processing Conference (EUSIPCO), 2017 25th European, p. 2581. doi: 10.23919/EUSIPCO.2017.8081677.

[11] Salomon, D., Motta, G. "Handbook of Data Compression" Fifth edition, Springer, 2010

[12] Shen, H., Pan, W. D. and Wu, D. (2017) 'Predictive Lossless Compression of Regions of Interest in Hyperspectral Images With No-Data Regions', IEEE Transactions on Geoscience & Remote Sensing, 55(1), pp. 173–182. doi: 10.1109/TGRS.2016.2603527.

[13] Huffman, D. 'A Method for the Construction of Minimum-Redundancy Codes' (1952) Proceedings of the IRE, Proc. IRE, (9), p. 1098. doi: 10.1109/JRPROC.1952.273898.

[14] Wallace, G.K. "The JPEG still picture compression standard" (1992) IEEE Transactions on Consumer Electronics, Consumer Electronics, IEEE Transactions on, IEEE Trans. Consumer Electron, (1). doi: 10.1109/30.125072.

[15] Abramson, N. (1963) Information theory and coding. McGraw-Hill (McGraw-Hill electronic sciences series).

[16] Chollet, Y. Finite state entropy, Viewed 5/5-19 https://github.com/Cyan4973/FiniteStateEntropy

[17] Storer, J.A. and Szymanski, T.G. (1982) "Data Compression via Textual Substitution," Journal of the ACM, 29:928–951.

[18] Ziv, J and Lempel, A. (1977) "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, IT-23(3):337–343.

[19] Sayood, K. "Introduction to Data Compression" Third edition, Morgan Kaufmann, 2006

[20] Welch, TA. "A Technique for High-Performance Data Compression" (1984) Computer, (6), p. 8. doi: 10.1109/MC.1984.1659158.

[21] Ziv, J and Lempel, A. (1978) "Compression of Individual Sequences via VariableRate Coding," IEEE Transactions on Information Theory, IT-24(5):530–536.

[22] "C++ reference", Viewed 14/5-19, https://en.cppreference.com/

[23] Mogahed, H., Yakunin, A. "Development of a Lossless Data Compression Algorithm for Multichannel Environmental Monitoring Systems" (2018) 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE), Actual Problems of Electronics Instrument Engineering (APEIE), 2018 XIV International Scientific-Technical Conference on, p. 483. doi: 10.1109/APEIE.2018.8546121.

[24] Spiegel, J., Wira, P. and Hermann, G. "A Comparative Experimental Study of Lossless Compression Algorithms for Enhancing Energy Efficiency in Smart Meters" (2018) 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), Industrial Informatics (INDIN), 2018 IEEE 16th International Conference on, p. 447. doi: 10.1109/INDIN.2018.8471921.

[25] Pope, J., et al. "An accelerometer lossless compression algorithm and energy analysis for IoT devices" (2018) 2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW), Wireless Communications and Networking Conference Workshops (WCNCW), 2018 IEEE, p. 396. doi: 10.1109/WCNCW.2018.8368985.

[26] Le, T L. and Vo, M-H. "Lossless Data Compression Algorithm to Save Energy in Wireless Sensor Network" (2018) 2018 4th International Conference on Green Technology and Sustainable Development (GTSD), Green Technology and Sustainable Development (GTSD), 2018 4th International Conference on, p. 597. doi: 10.1109/GTSD.2018.8595614.

[27] Nieminen, J., "An efficient LZW implementation", Viewed 20/5-19 http://warp.povusers.org/EfficientLZW/index.html

[28] Adler, M. "Zlib implementation", Viewed on 20/5-19 https://github.com/madler/zlib

Printed by Tryckeriet i E-huset, Lund 2019

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-710
http://www.eit.lth.se