
This is the **published version** of the master thesis:

Khoirunisya, Lina; Serra-Sagristà, Joan, dir. The Compression of IoT Operational Data Time Series in Automatic Weather Station (AWS). 2020. 67 pag. (1170 Màster Universitari en Enginyeria de Telecomunicació / Telecommunication Engineering)

This version is available at <https://ddd.uab.cat/record/259529>

under the terms of the  license



A Thesis for the
Master in Telecommunication Engineering

The Compression of IoT Operational Data Time Series in Automatic Weather Station (AWS)

by
Lina Khoirunisya

Supervisor: Joan Serra-Sagristà

Department of Information and Communications Engineering

**Escola d'Enginyeria (EE)
Universitat Autònoma de Barcelona (UAB)**

Bellaterra, June 2020



El sotasignant, *Joan Serra-Sagristà*, Professor de l'Escola d'Enginyeria (EE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Treball Final de Master ha estat realitzat sota la seva direcció per l'alumne *Lina Khoirunisya*.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, *26 de Juny de 2020*.

Signatura: *Joan Serra-Sagristà*

Resum:

Aquest projecte examina la compressió d'algorismes per a dades de series temporals d'Automatic Weather Station (AWS). El principal objectiu del projecte és obtenir una solució de compressió eficient. S'analitza el rendiment de diversos algorismes de compressió per reduir la quantitat de dades que es transmeten des del AWS al servidor. A més, s'ofereix informació sobre la comparació d'algorismes que són adequats per aplicar sobre les dades de AWS. Reduint la mida de les dades, podem disminuir també el cost de la transmissió. S'han comparat quatre algorismes, el Huffman Code, Arithmetic Coding, Lempel Ziv 7 (LZ77), i Lempel Ziv 4 (LZ4). Al final, la simulació i l'anàlisi s'han portat a terme basant-se en els resultats dels experiments fets al laboratori.

Resumen:

Este proyecto examina compresiones de algoritmos para datos de series temporales de Automatic Weather Station (AWS). El principal objetivo del proyecto es dar con una solución utilizando los datos del algoritmo de compresión para reducir la cantidad de datos que se necesitan para transmitir desde el AWS al servidor. Además ofrece información sobre la comparación de algoritmos que son adecuados para aplicar los datos de AWS. Reduciendo el tamaño de los datos, podemos disminuir el coste de la transmisión. Se han comparado 4 algoritmos, estos son Huffman Code, Arithmetic Coding, Lempel Ziv 7 (LZ77), y Lempel Ziv 4 (LZ4). Al final, la simulación y el análisis de han llevado a cabo basándose en los resultados de los experimentos hecho en el laboratorio.

Summary:

This project examines compression algorithms for time series data from Automatic Weather Station (AWS). The main goal of the project is to provide a solution using data compression algorithm to reduce the amount of data that needs to be transmitted from AWS to the server. Also to provide information about the comparison of algorithms that are suitable for application in AWS data. By reducing the size of the data, we can decrease the cost of transmission. Four compression algorithms are compared, those are Huffman Code, Arithmetic Coding, Lempel Ziv 7 (LZ77), and Lempel Ziv 4 (LZ4). At the end, the simulation and analysis have been carried out based on the results of experiments that have been done in the laboratory.

Acknowledgments

I like to express my sincere gratitude to the patrons of this work, without whom it was impossible for me to accomplish this onerous task.

I would first like to thank my supervisor prof. Joan Serra-Sagristà for always kindly helping me and giving so much guidance on my thesis. His encouragement helped in every stage of accomplishment of this work.

Thank you to my family, Siti Maryam, Imam Turidi, Nadia Isnaini Rahmah, and M. Rifki Cahaya Putra. To always supports me and provides unlimited motivation.

To my friends, those who always cheer me up: Paula Erill Roig, Laura Cortez, Brimoresa Wahyu, Nadia Safitri, Anindya Samhita, Laura Rincon, Serra Sensoy, and Lucie Robaye. All your love and advice means a lot to me.

To my bestfriends, Happy Fibi and Haris Hidayatulloh, for always be there.

Finally, special thanks to Diego Alexis and Ricardo Espejo for continuously encouraging me throughout years of my study. Thank you for your constant love.

Lina Khoirunisya
Bellaterra, July 2020

Table of Contents

1. Introduction.....	1
1.1. Background.....	1
1.2. Statement of the Problem	2
1.3. Objective.....	2
1.4. Methodology.....	2
1.5. Thesis Outline	4
2. Theory	5
2.1 Meteorology IoT	5
2.1.1 Automatic Weather Station (AWS).....	5
2.1.2 Time Series Data	7
2.2 Lossy Data Compression.....	10
2.3 Lossless Data Compression	11
2.3.1 Huffman Code.....	11
2.3.2 Arithmetic Coding.....	12
2.3.3 Lempel Ziv 7 (LZ77).....	13
2.3.4 Lempel Ziv 4 (LZ4).....	14
2.4 Visual Studio Code	15
2.5 Related Works.....	16
3. Design and Analysis	18
3.1 Experimental Design	18
3.2 Algorithm Research	19
3.2.1 Computational Difference Method.....	19
3.2.2 Huffman Code.....	20
3.2.3 Arithmetic Coding.....	24
3.2.4 Lempel Ziv 7 (LZ77).....	27
3.2.5 Lempel Ziv 4 (LZ4).....	29
4. Implementation	33
4.1 Implementation Results	33
4.1.1 Huffman Code	34
4.1.2 Arithmetic Coding	37
4.1.3 Lempel Ziv 7 (LZ77)	39
4.1.4 Lempel Ziv 4 (LZ4)	41
4.1.5 Comparison of 4 Algorithms.....	43
5. Conclusion and Future Work.....	47
5.1 Conclusion	47
5.2 Future Work.....	48
6. Bibliography	49
7. Appendix	50

List of Figures

Figure 1.1 Methodology of the project	2
Figure 2.1 Automatic Weather Station (AWS)	6
Figure 2.2 The communication of Automatic Weather Station (AWS)	7
Figure 2.3 Automatic Weather Station location	7
Figure 2.4 Complete data file	9
Figure 2.5 Specific data file	10
Figure 2.6 Example of Huffman Tree.....	12
Figure 2.7 The value of encoded character	12
Figure 2.8 Step of Arithmetic Coding algorithm	13
Figure 2.9 Step of LZ77 algorithm.....	14
Figure 2.10 Sequence of LZ4	15
Figure 2.11 Visual Studio Code	15
Figure 3.1 Input data.....	18
Figure 3.2 Example of computational difference method	19
Figure 3.3 Decoding process of Arithmetic Coding	26
Figure 3.4 LZ4 frame structure	29
Figure 3.5 LZ4 sequence structure	29
Figure 3.6 Linear Small Integer Code (LSIC) flowchart	30
Figure 3.7 Step of LZ4 compression	31
Figure 3.8 Deduplication of LZ4.....	31
Figure 4.1 Compression ratio of complete data (Huffman)	34
Figure 4.2 Temperature data (Huffman).....	35
Figure 4.3 Humidity data (Huffman).....	35
Figure 4.4 Wind speed data (Huffman)	36
Figure 4.5 Wind direction data (Huffman)	36
Figure 4.6 Compression ratio of complete data (Arithmetic)	37
Figure 4.7 Temperature data (Arithmetic)	38
Figure 4.8 Humidity data (Arithmetic)	38
Figure 4.9 Wind speed data (Arithmetic)	38
Figure 4.10 Wind direction data (Arithmetic).....	38
Figure 4.11 Compression ratio of complete data (LZ77)	39
Figure 4.12 Temperature data (LZ77)	40
Figure 4.13 Humidity data (LZ77)	40
Figure 4.14 Wind speed data (LZ77).....	40
Figure 4.15 Wind direction data (LZ77).....	40
Figure 4.16 Compression ratio of complete data (LZ4)	41
Figure 4.17 Temperature data (LZ4)	42
Figure 4.18 Humidity data (LZ4)	42
Figure 4.19 Wind speed data (LZ4).....	42
Figure 4.20 Wind direction data (LZ4).....	42
Figure 4.21 Compression ratio average of complete data.....	43
Figure 4.22 Compression ratio average of temperature data	44
Figure 4.23 Compression ratio average of humidity data.....	44
Figure 4.24 Compression ratio average of wind speed data	45
Figure 4.25 Compression ratio average of wind direction data	45

List of Tables

Table 1 Huffman character information	21
Table 2 Huffman code	24
Table 3 Arithmetic Coding data information	25
Table 4 Arithmetic Encoding Result	26
Table 5 LZ77 data information	27
Table 6 LZ4 character information.....	31
Table 7 Input data	31
Table 8 File size of input data	34
Table 9 Redundancy of simulation (Arithmetic).....	38

1. Introduction

The Internet of Things (IoT) has now become an important part of our lives. Hundreds of devices are capable of interacting with each other through wireless connections. IoT is expected to be one solution to be able to apply the concept of smart life. In 2020, the estimated number of IoT devices that have been installed is 28 billion [1]. With the addition of the amount of data transmission usage, data compression on the IoT devices becomes an important part. Data compression is the process of modifying, encoding, or converting the structure of the bit of data to reduce the size [2].

The goal of this project is to define and evaluate a solution for transmitting time series data from an Automatic Weather Station (AWS) system. This first chapter describes the background behind the project, the context of the problem, the objective of the project, the methodology, and the outline of the thesis.

1.1 Background

The concept of IoT has also been applied to the field of meteorology. An automatic weather station is the main equipment for automatic surface observation. This station provides information on weather and transmits or records observations obtained from measuring instruments. Most of the AWS consist of a data logger, rechargeable battery, telemetry, and meteorological sensors.

The sensor will record all information and store it in the data logger, then transmit it in real-time through wireless networks to the server, consisting of temperature, relative humidity, wind speed, and all the information of weather. The data logger manages the communication protocols with the remote server. This tool is placed in strategic locations to find out weather information in the area. In Barcelona, more than 100 AWS have been installed. All data will be transmitted in real-time to provide accurate results because the weather is always changing every time.

1.2 Statement of the Problem

Data loggers generally use a 3rd Generation (3G) and 4th Generation (4G) cellular network to transmit the data. The problem refers to the huge amount of data that needs to be transmitted from AWS to the server. It requires a high-cost service and huge memory storage. With tens of AWS located in Barcelona, huge amounts of data need to be transmitted continuously.

1.3 Objective

The main objective of the project is to provide a solution using data compression algorithm to reduce the amount of data that needs to be transmitted from Automatic Weather Station (AWS) to the server. Also to provide information about the comparison of algorithms that are suitable for application in AWS data. Alternatively, if no solution exists, then changes to the requirements will be suggested that could enable a solution.

1.4 Methodology

The methodology used to obtain the final results in the present work is divided into 8 parts, has been the following:

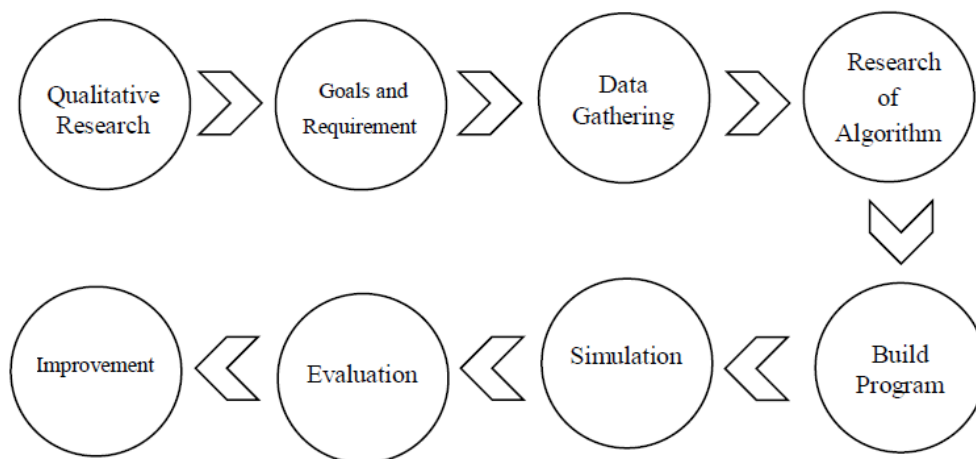


Figure 1.1. Methodology of the project

1. Qualitative Research

Research has been done by conducting a literature review with collecting information from the journal, book, and conference paper regarding data compression, also analyzed

the data based on the method and its application. The last process was to implement the technique using computer programming. This part is to give a clear idea about the theory and application of data compression.

2. Goals and Requirement

Goals and requirements need to be set to make experiments works optimally. This is needed due to limited time of the thesis process.

3. Data Gathering

Times series data will be obtained from AWS and consist of temperature, relative humidity, and wind. The AWS is located in Barcelona, Spain.

4. Research of Algorithm

From the data collection, it will be decided which algorithm is the most effective for data compression. In this section, we have selected the four most suitable algorithms for compressing data from AWS.

5. Build Program

When the algorithm has been decide, the next step is to make the computer program. The computer program will be build with C++ language using Visual Studio Code application. The program should be able to compress the original data before it transmitted to the server.

6. Simulation

The simulation will be conducted when all the data and computer program is created. This process will give us the result of the experiment. In this case, we can still improve the performance of the program.

7. Evaluation

The analysis will be done based on the compression ratio and the computational complexity of the technique. In this section, a comparison of the techniques will be discussed.

8. Improvement

After the evaluation, there will be a follow up to improve the experiment or suggestion for the next work.

1.5 Thesis Outline

The outline of the project will be divided into five chapters accordingly to the development process.

The chapter 1 shows the introduction, the problem statement, and the objective of the thesis. Also explains the methodology used in this project.

In chapter 2 shows some theoretical concepts about data compression concept and data compression techniques which are the basis of this project. In data compression techniques section, four types of techniques will be introduced.

Chapter 3 is focused on the design and analysis of the proposed system. This section explains the results obtained in simulation stage. The simulations have been carried out using the Visual Studio Code application.

Chapter 4 intends to depict an analysis of the result data obtained from the simulation. The data compression ratio and comparison among the techniques are shown.

Chapter 5 describes the conclusions of each stage along the project.

2. Theory

2.1 Meteorology Internet of Things (IoT)

The application of IoT in the field of meteorology became one of the discoveries that had a significant impact. The weather changes all the time, so it needs a device that can always update the weather data. The weather sensors produce data and gather information to be sent to various destinations that might need insight into meteorological trends. These include the airline company, news stations, event companies, logistics companies, and many others.

2.1.1 Automatic Weather Station (AWS)

Automatic Weather Station takes important place to provides information of the weather. It is an instrument that measures and records meteorological parameters using sensors without intervention of humans. Automatic Weather Station consists of several components:

- **Solar Panel**

Solar energy is the reliable source of energy since most of AWS are located at remote stations. Solar panel provide power to run the devices in AWS.

- **Battery**

Batteries store the energy generated by the solar panels and ensure functioning when solar energy is not available.

- **Sensors**

Some sensors that usually placed on AWS consists of Anemometer to measure the velocity of the wind, Thermometer are used to measure the temperature of the photovoltaic panels, Wind Vane to record the direction of the wind, Hygrometer are used to measure humidity, Rain Gauge to find out the rate of rainfall, and Barometer to measure atmospheric pressure at the given location.

- **Data Logger**

Data Logger is the core of every measurement station and perform key tasks. The main functions of a data logger are measurements, calculations (average, minimum, etc.), data storage, power supply, and communications. The system may report in near real time or

save the data for later recovery. Telemetry is usually attached in data logger and it is used to transmit data from AWS to the server.

- Enclosure

In extreme weather areas waterproof box should be rust proof and salt resistant. These enclosures should be made of suitable material or properly shielded so that inside temperature does not increase considerably, causing malfunction to the electronic equipment or batteries.



Figure 2.1. Automatic Weather Station (AWS)

In Barcelona there are hundreds of weather stations that operate and update data every time. Sensors that have been installed on AWS will collect data according to their respective functions. The data will be stored in a data logger in the form of float-point numbers (explain in Time Series Data part). By using a transmission network, all data will be sent to the server. Data loggers generally use a 3rd Generation (3G) and 4th Generation (4G) cellular network to transmit the data. Central server will distribute the data to the client and monitor the weather conditions. Currently there are many websites that provide accurate weather data globally.

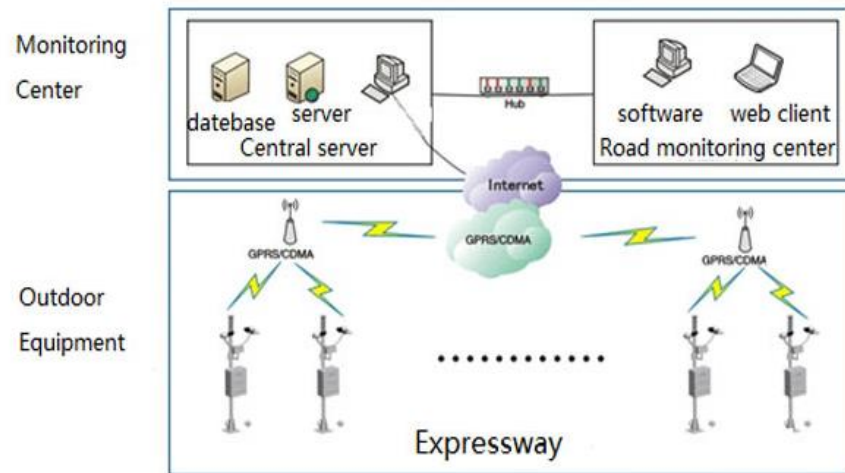


Figure 2.2. The communication of Automatic Weather Station (AWS)

In this project, there are 6 AWS used as references and data sources, all located in Barcelona. From Figure 4 we can see the red point represent the location of every AWS in this project. The list of the AWS are:

- Meteoblue Eixample
- Weathercloud Sant Pol de Mar
- Weathercloud Institut Europa
- Barcelona El Raval
- Observatori Fabra
- Barcelona Zona Universitaria

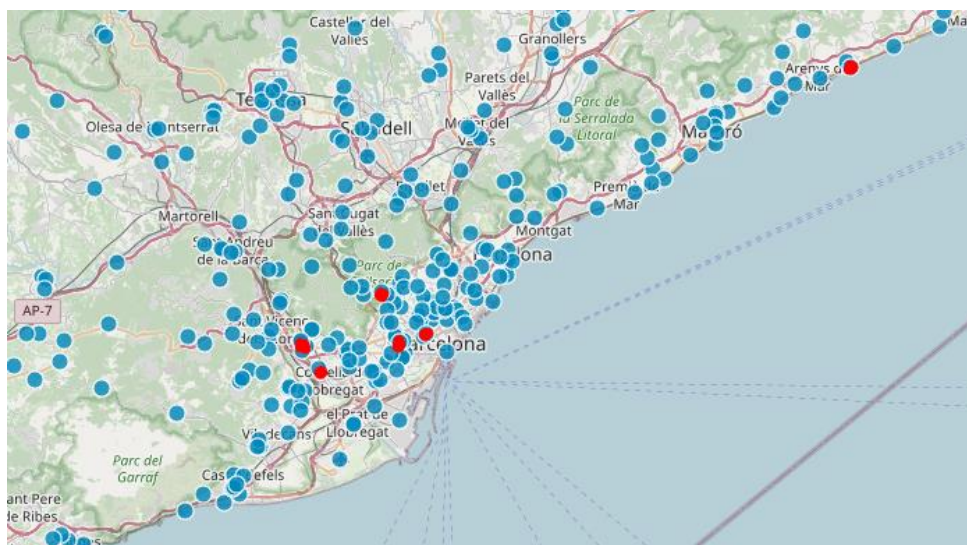


Figure 2.3. Automatic Weather Station location

2.1.2 Time Series Data

From data logger, we will obtain the time series data. The contents of the information and the time span of every AWS are varied, this is due to the different AWS location and different AWS type, some AWS provide more sensor than other AWS.

The sources of the data are some sensors mounted on AWS with the format of most raw data in time series, containing timestamps and values. The value of time series data is all numeric, i.e., none of these values are text or characters. Timestamps are represented as integer numbers, while values are mostly float-point numbers, whose properties depend on the source of data. In this matter, we changed all the data into an integer form. The floating-point data is not friendly to be applied in data compression, because it cannot be represented exactly as a binary floating-point value, but rather as a bit value. We can change all the data in this project into integer form because we know exactly how decimal points of precision each variable has.

This section explains the source and contents of the file that will be compressed. All data obtained from Automatic Weather Station (AWS) spread throughout Barcelona. The detail of each station are:

1. Meteoblue Eixample

The data obtained from a website that provides information about the weather (<https://www.meteoblue.com/>). Location of the AWS is in Eixample, Barcelona with coordinate of 41.3°N/ 2.16°E. The time span of data is 1 year and 3 months with a time difference of every hour. The contents of the file consist of temperature, humidity, wind speed, wind direction, sun shine, snowfall, rain rate, and cloud cover.

2. Weathercloud Sant Pol de Mar

Weathercloud is a website who provides weather information from all over the world (<https://weathercloud.net/>). AWS Sant Pol de Mar is located in Sant Pol de Mar, Barcelona, with coordinate of 41.36°N/ 2.37°E. The time span of data is 2 years with a time difference of every ten minutes. The contents of the file consist of temperature, humidity, wind speed, wind direction, rain rate, and heat rate.

3. Weathercloud Institut Europa

This AWS located in L'Hospitalet de Llobregat, Barcelona, with coordinate of 41.21°N/ 2.6°E and managed by Institut Europa. The time span of data is 2 years with a time

difference of every one day. The contents of the file consist of temperature, humidity, wind speed, rain rate, sun radiation, wind direction, and wind speed.

4. Barcelona El Raval

From the website of city hall of Barcelona (<https://opendata-ajuntament.barcelona.cat>) we obtained data for the El Raval area, with coordinate of 41.22°N/ 2.10°E. The file consists of data of temperature, humidity, rain rate, wind speed, wind direction, air pressure, and cloud information. The time span of data is 10 years with a time difference of every one day.

5. Observatori Fabra

The data obtained from the city hall Barcelona database. This AWS is managed by observatori Fabra located in Sarria, Barcelona, with coordinate of 41.41°N/ 2.13°E. The content of the file consists of data of temperature, humidity, rain rate, wind speed, wind direction, air pressure, and cloud information. The time span of data is 10 years with a time difference of every one day.

6. Barcelona Zona Universitaria

From the database of city hall Barcelona, we managed to obtain AWS data for Zona Universitaria. Located in coordinate of 41.22°N/ 2.06°E. The file consists of data of temperature, humidity, rain rate, wind speed, wind direction, air pressure, and cloud information. The time span of data is 10 years with a time difference of every one day.

There will be two different type of data we used in the simulation. Those are ‘complete data file’ consists of all original data from the AWS. And ‘specific data file’, where we simulate one type of data at a time.

Complete data file is a file that contains all the original information obtained from AWS. Data is written in Ms. Excel (.xlsx) but it can not be processed perfectly by the computer program, so all data in this simulation must be converted to a text file (.txt) and changed into integer form.

Year	Month	Day	Hour	Minute	Temperature [2 m above gnd]	Relative Humidity [2 m above gnd]	Wind Speed [10 m above gnd]
2019	1	1	0	0	11642492	95 77668280	84093850 10198 0 0 0 0 43366780 31251044 13797913 28512402
2019	1	1	1	0	11422492	95 70382050	60524117 10194 0 0 0 0 49121733 31060132 15281989 28501837
2019	1	1	2	0	11402493	94 75443420	47862396 10197 0 0 0 0 60748550 30787500 17377226 28317255
2019	1	1	3	0	11502492	92 81022390	46332200 10190 0 0 0 0 68231690 30865982 19296133 28403625
2019	1	1	4	0	11372492	91 84788885	43726960 10176 0 0 0 0 69926825 31036456 18000000 28626020
2019	1	1	5	0	11252492	89 81546260	38367477 10177 0 0 0 0 67814364 31500000 16595179 29434110
2019	1	1	6	0	11262492	85 70331616	37303940 10177 0 0 0 0 62419515 32019443 15003839 30025644
2019	1	1	7	0	11182492	85 60397053	41423660 10172 0 0 0 0 57012090 32259467 12599998 30686990
2019	1	1	8	0	10912492	86 58426080	46847595 10176 0 0 0 0 53941890 32709480 8161764 31142368
2019	1	1	9	0	11242492	84 64488610	51709840 10174 0 0 0 0 20288420 29319860 38268526 31118590
2019	1	1	10	0	11822492	83 92667380	71565040 10172 0 0 0 0 28692157 20180140 25455842 18813010

Figure 2.4. Complete data file

Beside of processing data from AWS as a whole, the compression process is also completed using specific data. The specific data will be processed in this project are data of temperature, humidity, wind speed, and wind direction. The reason to select these data from the data logger is that this information is all collected from the sensor attached in AWS, all four sensors are the main sensor of a weather station.

Temperature	A degree of hotness or coldness the can be measured using a thermometer.
Humidity	The concentration of water vapour present in the air.
Wind Speed	Fundamental atmospheric quantity caused by air moving from high to low pressure.
Wind Direction	Wind direction is reported by the direction from which it originates.

```
Tempin
207
207
208
210
211
213
214
215
216
217
```

Figure 2.5. Specific data file

2.2 Lossy Data Compression

Lossy data compression is a compression method that does not decompress data back to the original form. After the application of a lossy compression algorithm, the original data can not be reconstructed from the compressed data. Some contents and parts may be lost. But with this technique, we can get a high compression ratio. If we expect to get high compression capability without regard to decompression results, then this technique is very suitable.

Lossy data compression is widely used in JPEG images, MPEG video, and MP3 audio formats. One of the most common uses is in image compression. This method will degrade the image quality to meet a given target data rate for storage and transmission. It will select only the main pixel of the image, thus producing an image that is similar to the original image.

2.3 Lossless Data Compression

Lossless data compression allows the original data to be exactly recovered from their compressed form. When we need certainty that we achieve the same what we compressed after decompression, lossless compression methods are the only choice. The thesis will only focus on the lossless compression because the data to be sent must be perfectly readable again by the recipient and because of the importance of trace data integrity. The idea of all lossless compression is to extract a pattern from the data, giving greater compression to the most frequently appearing patterns, so that the total size shrinks while no information is lost. This provides a full reconstruction of the original data.

One of the performance indicators of a compression technique is data compress ratio. Compression ratio is the ratio between the size of the compressed file and the size of the source file. The formula to calculate the data is written in equation 1.

$$\text{Compression ratio} = (1 - \frac{\text{size after compression}}{\text{size before compression}}) \times 100\% \quad (1)$$

2.3.1 Huffman Code

Huffman code is based on representing the symbols that have high occurrence probabilities with shorter code words and assigning longer code words to symbols that have low occurrence probabilities. It is an entropy coding technique published by David A. Huffman in 1952 [3]. The process consists of three steps. Those are calculate the probabilistic distribution of the character, then create a binary tree. The leaves represent the symbols of the input file. The code length for these symbols equals their depth in the tree, so it will be their distance to the root node. And lastly, find the unencoded characters in the tree and encode them as the path from the root to the leaf.

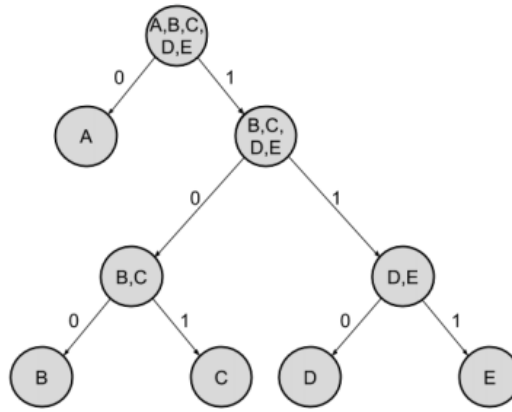


Figure 2.6. Example of Huffman Tree

A	0
B	100
C	101
D	110
E	111

Figure 2.7. The value of encoded character

The Huffman coding is simple, even though rebuilding the tree after processing each character is quite complicated. The loss of this algorithm is relatively small, all because of its simplicity and effectiveness. This is very appropriate if we want to prioritize the speed of compression. Gallager[7] shows that the maximum inefficiency is bound by equation 2. Some of the inefficiency are the maximum difference between the expected code length and the optimum. From equation 2, P_m is the probability of occurrence of the most frequent symbol.

$$P_m + \log \frac{2 \log e}{e} \approx P_m + 0.86 \quad (2)$$

2.3.2 Arithmetic Coding

Arithmetic coding is one of the most optimal entropy coding techniques if the objective is the good compression ratio [8], but it also has the most complicated complexity. This algorithm was developed by IBM Company in 1979. Arithmetic coding can achieve a better compression ratio compared with Huffman coding. It is because Arithmetic coding uses fractional bits for its code words, while Huffman coding uses an integral number of bits. It does not produce a

single code for each symbol. Instead, it produces code for an entire message by incrementally modifying the output code.

Arithmetic coding transforms the input data into a single rational number between 0 and 1 [8]. For each sequence symbol, the current interval is divided into subintervals of length proportional to the frequencies of character occurrences. Then the subinterval of the current symbol is chosen again. This procedure is repeated for all characters from the input sequence. In the end, we output the binary representation of any number from the final interval. Some of the difficulties in Arithmetic Coding is it requires more CPU power and the shrinking of the current interval requires the use of high precision arithmetic. The encoding process in Arithmetic code technique explain in figure 2.8.

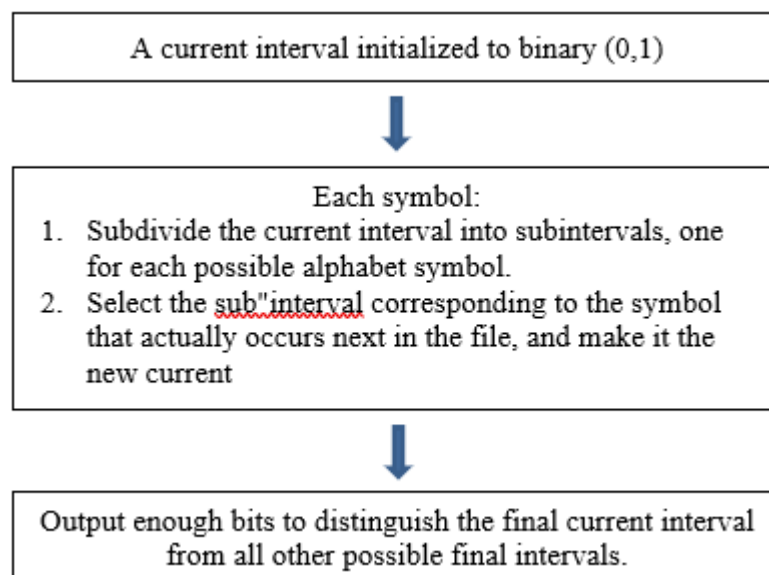


Figure 2.8. Step of Arithmetic Coding algorithm

2.3.3 Lempel Ziv 7 (LZ77)

Lempel Ziv 7 or what is commonly referred to LZ77 works by examining the input using a sliding window technique with a fixed size. The window consists of two parts, search buffer, and look-ahead buffer.

The left part of the sliding window is a search buffer, which includes the symbols that have been input and encoded recently. The length of the search buffer is equal to the length of the

dictionary, and it will be changed dynamically with the movement of the sliding window. The right part is a look-ahead buffer, it processes the input stream to be encoded. The length of the look-ahead buffer is equal to the value of the maximum length of the identical symbols. The distance between the selected match and the start of the look-ahead buffer is called offset.

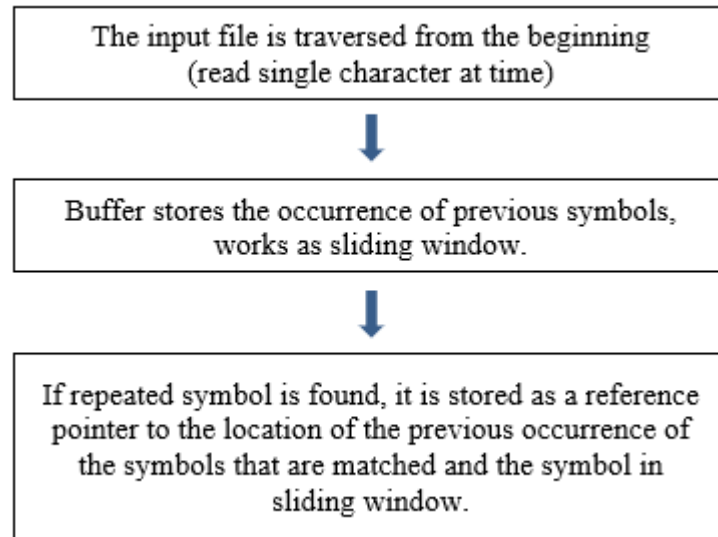


Figure 2.9. Step of LZ77 algorithm

The result of compression process is represented as distance that indicates the offset from the start of the match symbols found in the sliding window to the current symbol, next symbol indicates that new phrase was found, and length indicates the length of the match symbols.

2.3.4 Lempel Ziv 4 (LZ4)

LZ4 is a member of the LZ77 family, where this algorithm is focused on speed by lowering the compression ratio [9]. LZ4 adds to the base LZ77 algorithm by using more complex output codes that can better handle various cases. In the implementation, we have preference options that allow for varying compression efficiency.

The compressed data format of LZ4 is defined by the structure of block which consists of sequences. Each sequence starts with a token of one-byte value, separated into two 4-bits fields. It will provide the value range of 0-15. 4 high-bits of a token are used in the first field, and it will indicate the literal length. Following by literals, it is the uncompressed bytes to be copied as-is. The value of literals is exactly the same as the literal length. The next structure is offset,

this is a 2 bytes values. The function of offsite is to represent the position of the match to be copied from. The final structure is match length, where we use the second token field consists of 4 bits value, with a range value of 0 - 15. The structure is shown in Figure 2.10.

Token	Literal Length	Literals	Offset	Match Length
1 bytes	0 - N bytes	0 – M bytes	2 bytes	0 - N bytes

Figure 2.10. Sequence of LZ4

After obtained offset and match length, decoder will proceed to copy the repetitive data from the already decoded buffer. The process will start another one when it finished decoding the match length. We also need to pay attention on the overlapped copy. It is a condition where match length is bigger than offser, typically when there are numerous consecutive zeroes.

2.4 Visual Studio Code

Visual studio code is a source code editor for several programming language, such as C++, C#, Java, Phyton and PHP. This application is available for Windows, macOS, and Linux. In this project we use C++ programming language to build the program and windows as the computer system. C++ language was chosen because it is a powerful, efficient and it is widely used.

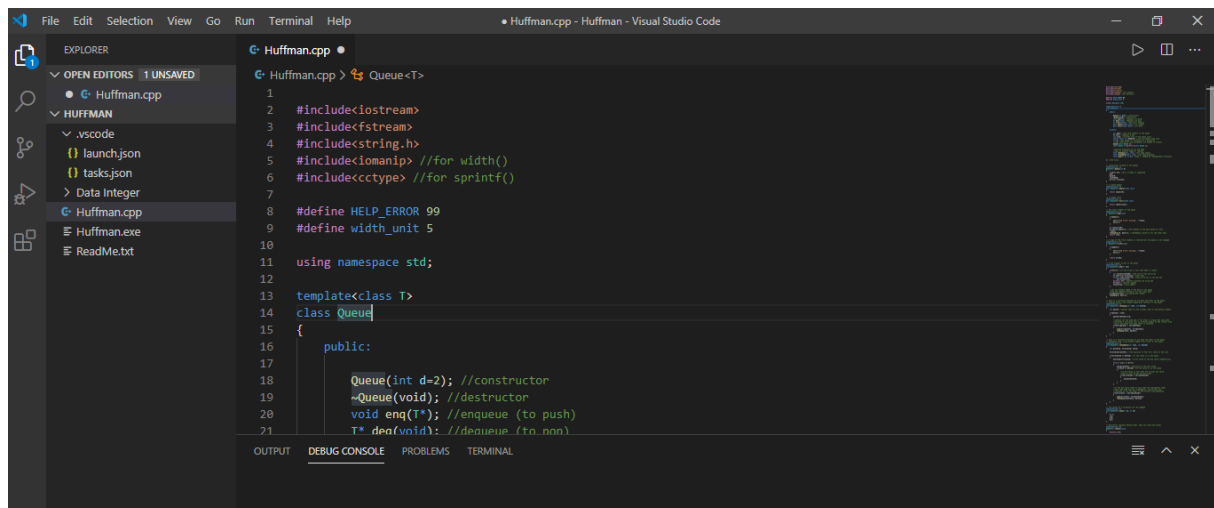


Figure 2.11. Visual Studio Code application

2.5 Related Works

There are many existing works related to time series compression with various methods and applications on IoT devices. Lossless algorithms are preferred on compressing data for real time database. That is because this algorithm offers a full reconstruction of the compression process, although it cannot provide a compression ratio at the level of lossy compression.

Fredrik Bjärås in his paper explain about the comparison of compression algorithms on time series data for IoT devices [10]. His experiment evaluates the performances of several algorithms by examining the combination of ratio, memory usage, and speed. Renzhi Xing also conducted an experiments of compression of IoT operational data time series in vehicle embedded system [11]. Where to compress the vehicle embedded system data, the best method is combined compression, which is the combination of lossy and lossless compression. This can be understood because the data to be sent in the project does not need to be reconstructed perfectly. In her experiment, only the statistical results of the data to be transmitted to the server.

Peter Zaitsev and Vadim Tkachenko [12] evaluated common lossless database compression algorithms regarding compression speed, compression ratio, and other relevant factors. From their experiments, the recommendation method are LZ4 and Snappy because of their fast compress/decompress speed and decent compression ratio.

The investigation about lossy and lossless compression on the electrodiagram (ECG) were also carried out by Chacko John, Chun Hueng, and Yong Lian [13]. The data is compressed using a lossy compression technique with a high compression and using lossy data with entropy coding. From the application in ECG, the scheme achieves an overall lossless compression ratio of 2.1x and a lossy compression ratio of 7.8x. The proposed scheme can be implemented either in sensor software or on-chip hardware and can be extended to other physiological signals and wireless sensors. The benefits of the proposed scheme which are demonstrated in an FPGA prototype, shows low complexity and significant power reduction and therefore is highly suited for wireless wearable sensors.

James Pope, Antonis Vafeas, Atis Elsts, George Oikonomou, Robert Pienchocki, and Ian Craddock also provided a report of an accelerometer lossless compression algorithm and energy analysis for IoT devices, where they use the sensor as a representation of IoT device [14]. From

the experiment, it can be concluded that the energy cost to compress data is less than energy to transmission data. It means the compression process saved more energy than the transmission process. An investigation about real-time quality control data compression was also carried out by Simhadri Vadrevu and M. Manikandan, they proposed a new real-time quality-control data compression framework implemented in Arduino Duo with a 32-bit Atmel SAM3X8E ARM Cortex-M3 processor [15]. The experiment aims to reduce the amount of data that needs to be transmitted that can significantly reduce the power consumption of wireless modules in health monitoring devices. A new real-time quality control data compression framework is presented for maximizing the battery life of IoT and Smartphone-based PPG monitoring devices. The proposed quality control PPG signal compression algorithm achieves an energy saving between 83% and 92%.

3. Design and Analysis

3.1 Experimental Design

The purpose of the experiments is to see and compare the performance of the different compression algorithms, also to see if different types of data affect the compression performance. Some conditions are worth knowing before doing the simulation.

a. Test environment

The source code used in the experiments were all implemented in C++ programming language, using a Windows 10 64-bit system. With 6GB and Intel i5-3317U CPU. The application used to edit and debug the program is Visual Studio Code.

b. Experiment input

The input of simulation is divided into two types, those are ‘complete data file’ and ‘specific data file’. We apply both types of data in four algorithms, Huffman Code, Arithmetic Coding, LZ77, and LZ4. We will also use input data after applying the computational difference method (explain in Computational Difference Method part), this is a part of the compression process to reach a better compression ratio. The detail of the input data is explained in Figure 3.1.

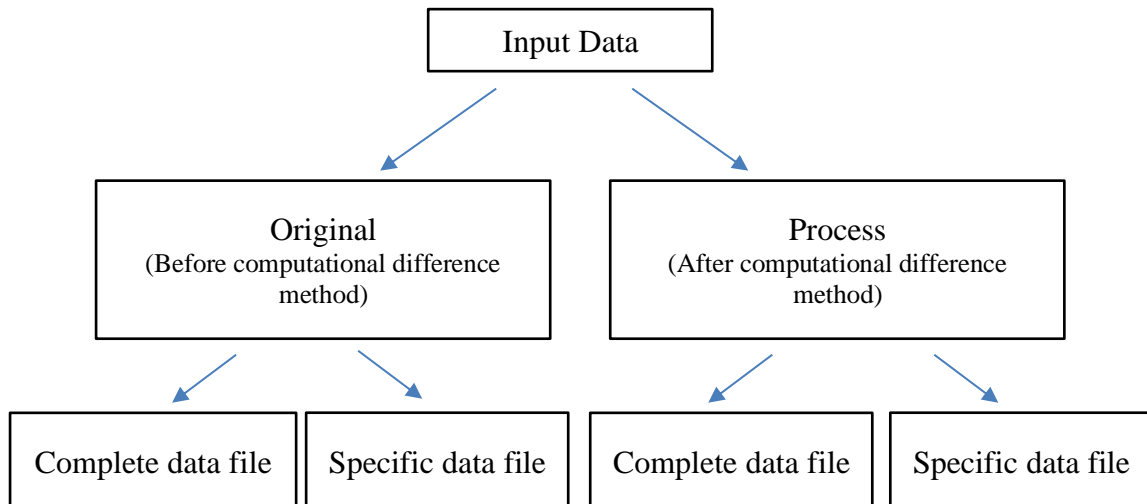


Figure 3.1. Input data

c. Experiment measurements

The experiment mainly measures the compression ratio, for both data types, ‘complete data file’ and ‘specific data file’. The compression ratio was defined as in section 2.3. The other comparison factor in this project is computational complexity.

3.2 Algorithm Selection and Design

In this section, we will discuss about four algorithms that has the potential as one of the best algorithms to be applied to time series data, those are Huffman Code, Arithmetic Coding, LZ77, and LZ4. But before we apply the algorithm, a computational difference method will be introduced to be able to simplify the process of compressing data.

3.2.1 Computational Difference Method

Computational difference method (CDM) is a process to take the values of two consecutive samples and calculate the difference between the first one and the next; the output is the difference. It is one of the step in the compression process to obtain a smaller size of the file, it also helps to perform a lossless compression achieving higher compression ratio. With this computation, we can reduce the entropy of the data. The example of the process shown in figure 3.2. To find how much effect this process has on compression data, the method will be applied to one of the main data input.

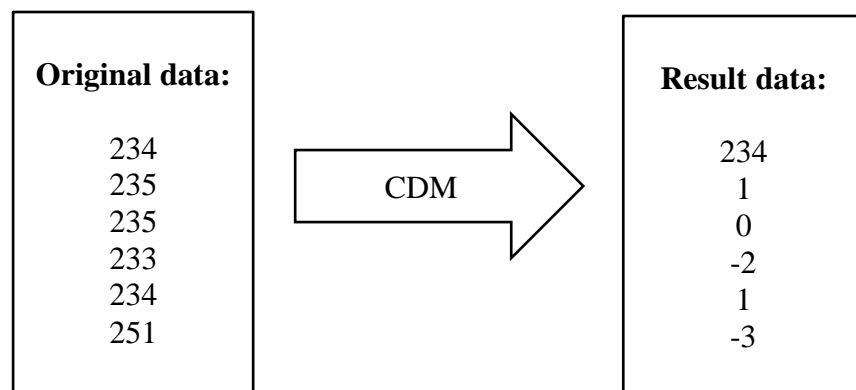


Figure 3.2. Example of computational difference method

3.2.2 Huffman Code

The general explanation of Huffman code has been discussed in 2.3.1. A Huffman tree representation is also related to average code length for a symbol in a message. Average code length can be defined as equation 3. Where F_i is the frequency of i^{th} symbol, P_i is the probability of i^{th} symbol, and l_i is the code length of i^{th} symbol.

$$\text{Average Code Length, ACL} = \frac{\sum_{i=1}^n F_i l_i}{\sum_{i=1}^n F_i} = \sum_{i=1}^n P_i l_i \quad (3)$$

To obtain the entropy of the message, we can use the equation 4.

$$\text{Entropy, H} = - \sum_{i=1}^n P_i \log_2 (P_i) \quad (4)$$

To calculate the total number of bits in Huffman encoded message, we can apply equation 5.

$$\text{Total Number of Bits, TNB} = \sum_{i=1}^n F_i l_i \quad (5)$$

The efficiency of Huffman coding in terms of size reduction is the result of difference of characters frequencies. Compression ratio decrease when this difference is less, it makes the probability of degeneration of Huffman coding is more. So compression ratio depends on standard deviation. Standard deviation for a symbol from its average code length can be defined by the following equation:

$$\text{Standard Deviation, SD} = \sqrt{\frac{\sum_{i=1}^n (ACL - l_i)^2}{n}} \quad (6)$$

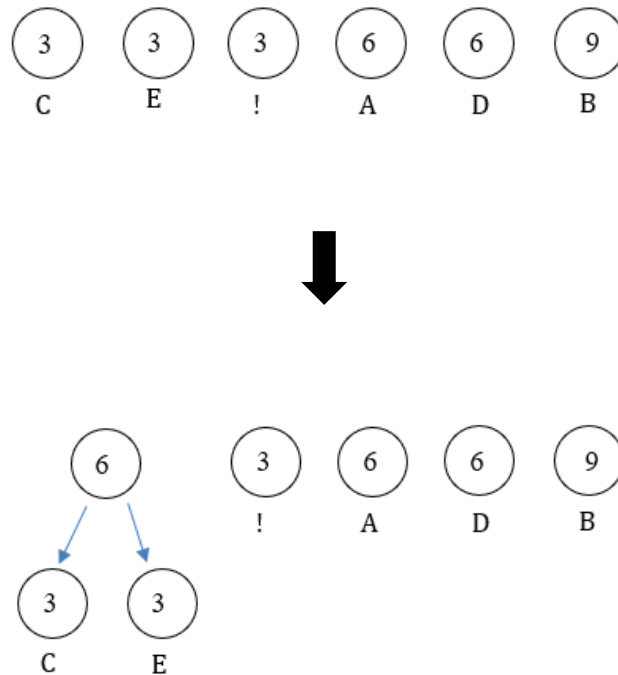
In its application, there are two major steps in Huffman Code, first is building a Huffman tree from the input characters. The second is assigning code to the characters by traversing the Huffman Tree. For example we have a data with detailed information as follows.

Character	Frequency
A	6
B	9
C	3
D	6
E	3
!	3

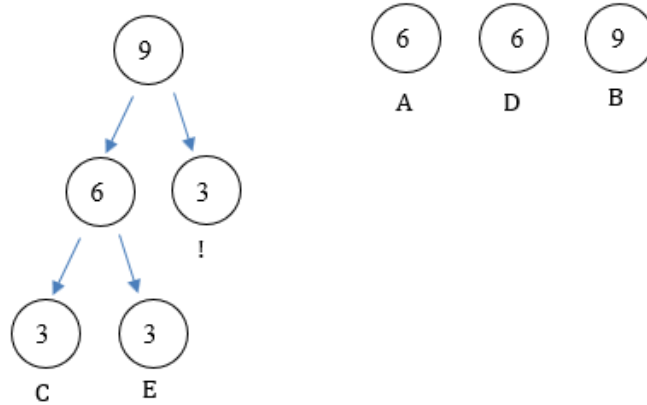
Table 1. Huffman character information

Table 1 shows the information of characters will be processed and how often these characters appear. To use Huffman code as data compression, we need to determine the Huffman code for each character, the average code length, and the length of Huffman encoded message. We can construct a Huffman tree in the following order.

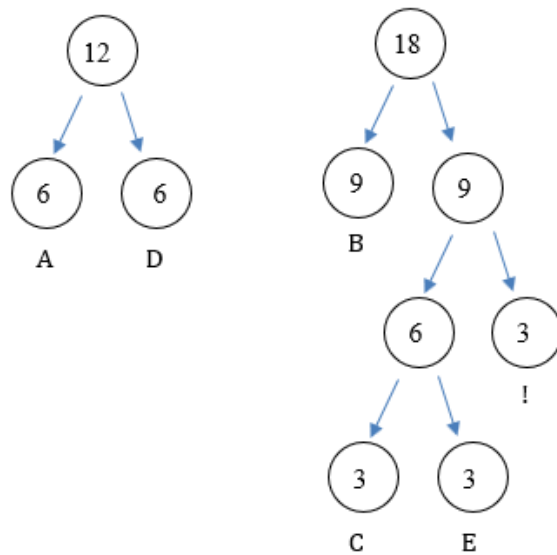
- a. Create a leaf node for each character of the text. It contains the occurring frequency of that character. Arrange all the nodes in increasing order of their frequency value.



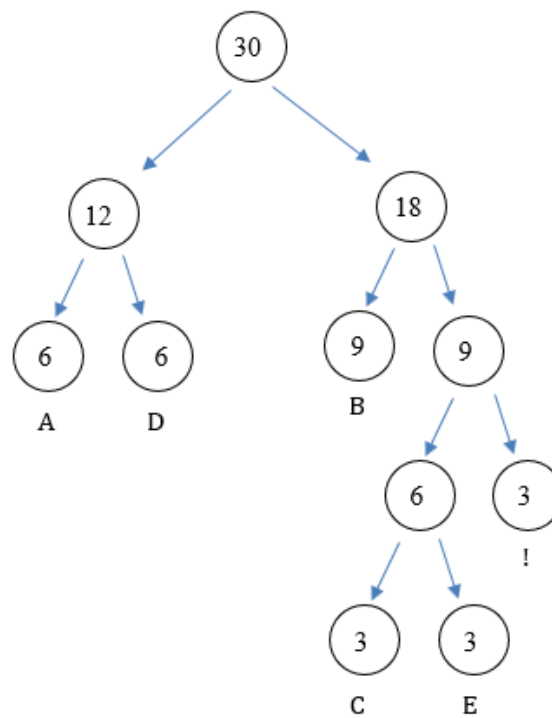
- b. Considering the first two nodes having minimum frequency, create new internal node. The frequency of the new node is the sum of two nodes frequencies. The first node will be a left child and the second will be the right child of the newly created node.



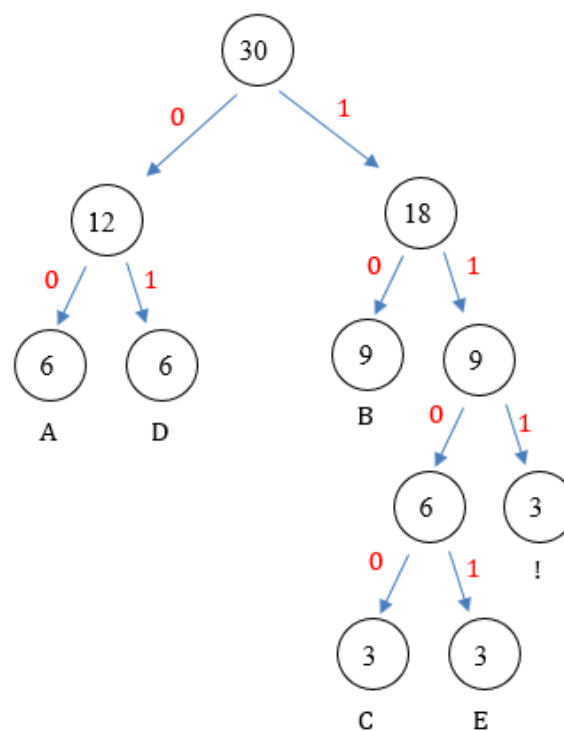
- c. We will do the same step until all the nodes form a single tree. The final tree is the desired Huffman tree.



The final result of Huffman Tree shows below.



- d. After we get the desired Huffman tree. We will assign weight to all the edges of the constructed Huffman tree. In this case, we will assign weight of '0' to the left edges and weight of '1' to the right edges.



- e. The Huffman code is written by combining information from the root node to the leaf node of the character. Table 2 shows the result of Huffman code. We can also conclude the character that occurs less frequently is assigned the larger code. And the character occurs more frequently is assigned the smaller code.

Character	Huffman Code
A	00
B	10
C	1100
D	01
E	1101
!	111

Table 2. Huffman Code

With all the information above, we can calculate the average code length using equation 3.

$$\begin{aligned}
 ACL &= \frac{\sum_{i=1}^n F_i l_i}{\sum_{i=1}^n F_i} \\
 &= \frac{(6 \times 2) + (9 \times 2) + (3 \times 4) + (6 \times 2) + (3 \times 4) + (3 \times 3)}{(6 + 9 + 3 + 6 + 3 + 3)} \\
 ACL &= 2.83 \text{ bits/symbol}
 \end{aligned}$$

To obtain the value of entropy, we can use the equation 4.

$$\begin{aligned}
 H &= - \sum_{i=1}^6 P_i \log_2 (P_i) \\
 &= - 1/\log_2 [(0.2 \log (0.2)) + (0.3 \log (0.3)) + (0.1 \log (0.1)) + (0.2 \log (0.2)) \\
 &\quad + (0.1 \log (0.1)) + (0.1 \log (0.1))] \\
 &= - 3.322 (-0.139 - 0.156 - 0.1 - 0.139 - 0.1 - 0.1) \\
 H &= 2.438 \text{ bits/symbol}
 \end{aligned}$$

We can see that the entropy value is smaller than the average code length. Where the difference is 0.392 bits/symbol.

3.2.3 Arithmetic Coding

Arithmetic coding is well known for its optimality, and the fact that it can be a very versatile and powerful tool for coding complex data sources [16]. In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. Frequently used characters will be stored when a string is converted to arithmetic encoding, with fewer bits and ‘not frequently’ occurring characters will be stored with more bits, resulting in fewer bits used in total. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows. In this matter, the successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model.

Before anything is transmitted, the first interval is $[0, 1]$ with a half-open interval of $0 \leq x < 1$. When the process of each symbol starts, the range is narrowed to that portion of it allocated to the symbol. For example we have an information about some alphabets presented in Table 3.

Symbol	Probability	Range
A	0.2	$[0, 0.2]$
B	0.3	$[0.2, 0.5]$
C	0.1	$[0.5, 0.6]$
D	0.2	$[0.6, 0.8]$
E	0.1	$[0.8, 0.9]$
!	0.1	$[0.9, 1.0]$

Table 3. Arithmetic Coding data information

A message of ‘BACC!’ will be transmitted. The first symbol is ‘B’, and the encoder narrow it into $[0.2, 0.5]$ based on the range on the table. The second symbol that is ‘A’, will narrow the new range to the first one-fifth of it. Because ‘A’ is allocated in the range of $[0, 0.2]$. The encoding process produces $[0.2, 0.26]$, since the previous range was 0.3 units long and one-fifth of that is 0.06.

The third symbol is 'C' and allocated in the range of [0.5, 0.6]. We will apply the same procedure as before to the [0.2, 0.26], so we will obtain the result of [0.23, 0.236]. The result of the next symbol, 'C', is [0.233, 0.2336] and symbol of '!' is [0.23354, 0.2336].

Symbol	Result
B	[0.2, 0.5]
A	[0.2, 0.26]
C	[0.23, 0.236]
C	[0.233, 0.2336]
!	[0.23354, 0.2336]

Table 4. Arithmetic encoding result

The decoder deduce the first symbol based on the final range [0.23354, 0.2336]. It will shows symbol of 'B' as the first character since the range lies entirely within the model of space Table 4 allocates for 'B'. Proceeding like this, the decoder can deliver the entire message.

With this process, the decoder could also face a problem. That is, if the process arrives at the end of the message, the decoder should know when the message has ended. To resolve the ambiguity, we use special End-of-File (EOF) symbol known by both encoder and decoder. If the decoder identifies the symbol, the process will automatically end. The decoder stops decoding. And the final results will be presented. In our case, the EOF symbol is '!'.

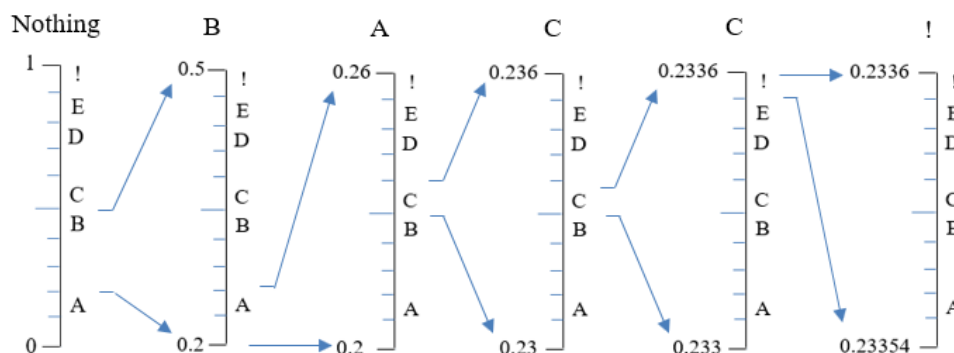


Figure 3.3. Decoding process of Arithmetic Coding

3.2.4 LZ77

LZ77 compression works by finding sequences of data that are repeated. The term ‘sliding window’ is used, it means at any given point in the data, there will be a record of the characters that have been processed. LZ77 iterates sequentially through the input string and stores any new match into a search buffer.

In the condition where the next character to be compressed is identical with one in the record list, the sequence of character is replaced by two numbers. Those are distance and length. A distance is the information of how far it backs to the window where the sequence starts, and a length represents the number of character for which the sequence is identical.

The compression process of LZ77 is divided into 3 step, first process is to find the longest match of a string in the window for the look-ahead buffer, then it will give output of a triple *o-l-c* (offset, length, and character), the last step is to move the cursor $l+1$ position to the right. A null pointer is generated as the pointer in case of absence of the match, and the first symbol in the look-ahead buffer (0,0,c).

Offset	It represents the position number in order to find the start of the matching strings.
Length	Represents the length of the match.
Character	The character found after the match, denotes the next symbol to be encoded


For example we have a data set shown in Table 5.

Input Data											
a	b	a	b	c	b	a	b	a	b	a	a

Table 5. LZ77 data information

We will start from the left part with an empty search buffer. The first data is ‘a’, with no match record in our search buffer. It gives us the result of *o-l-c* as (0,0,a), since we are not moving backwards. We will move the cursor to $l+1$ position, where we can get the character of ‘b’.


Input Data											
a	b	a	b	c	b	a	b	a	b	a	a



LZ77 encoding:
(0,0,a), (0,0,b)

Since we also have no record of 'b', the encoding result will be similar to the first character, that is (0,0,b). At this moment, in the search buffer we can find the record of 'a' and 'ab', but not 'abc'. So, the next character we can find is 'c' with output of $o-l-c$ is (2,2,c). It is because we need to move the position two times to the left ($o = 2$), and read two characters ($l = 2$).


Input Data											
a	b	a	b	c	b	a	b	a	b	a	a



LZ77 encoding:
(0,0,a), (0,0,b), (2,2,c)

The next character to be processed is 'b'. In the search buffer, we have the record of 'ba', 'bab' but not for 'baba' where this 'b' is located. So we need to put this information into the dictionary. We move the position 4 times to the left ($o = 4$), and read 3 characters ($l = 3$). The next character we can find is 'a', it gives us the encoding result of (4,3,a). The last two characters in the data also need to be processed with the same way.

Input Data											
a	b	a	b	c	b	a	b	a	b	a	a



LZ77 encoding:
(0,0,a), (0,0,b), (2,2,c), (4,3,a), (2,2,a)

The process can give us a high compression ratio. But if we notice, the time complexity of the algorithm does not seem to be the best. If all the character in the input data is different, we would need to process lot of data one by one. The time complexity of the last position of the data is shown by equation 7.

$$\text{Time Complexity, } O(n^2) = \frac{n(n-1)}{2} \quad (7)$$

3.2.5 LZ4

The LZ4 algorithm is focused on provides high speed by lowering the compress ratio. The important design principle behind LZ4 has been simplicity. It is known for its easy code and fast execution. To be able to find out more about this algorithm, it is important to know how the integer encoder works.

LZ4 is build by a structure which consists of frames, shows by Figure 3.4. And the frames composed from a magic number, frame descriptor, data blocks, end mark and content checksum.

Magic Num.	Descriptor	Data Block	(...)	End Mark	Content Cheksum
4 bytes	3-15 bytes	Upto 4MB		4 bytes	0-4 bytes

Figure 3.4. LZ4 frame structure

The stream of LZ4 is divided into some segments called blocks. The compressed block is composed of sequences, and each sequence starts with a *token*, which is a one-byte field separated into two 4-bit fields. The higher field t_1 is used for storing the length of literals. The sequence structure showed by Figure 3.5.

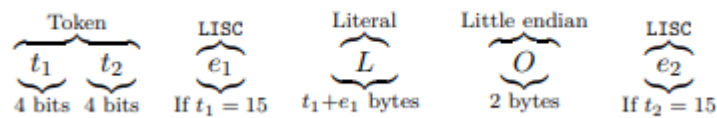


Figure 3.5. LZ4 sequence structure

For values higher than 15 it uses a Linear Small Integer Code (LSIC), it loads another byte after token adds it to the length and checks if the value of the read byte is equal to 0xFF. If this byte is the maximum value (255), another byte is read and added to the sum. This process is repeated until a byte below 255 is reached, which will be added to the sum, and the sequence will then end. In simple words, this process will continue to add bytes until it reaches a non 0xFF byte. All the processes are presented in Figure 3.6.

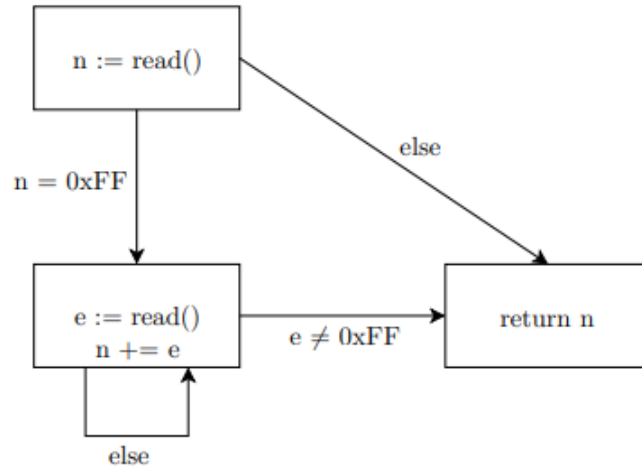
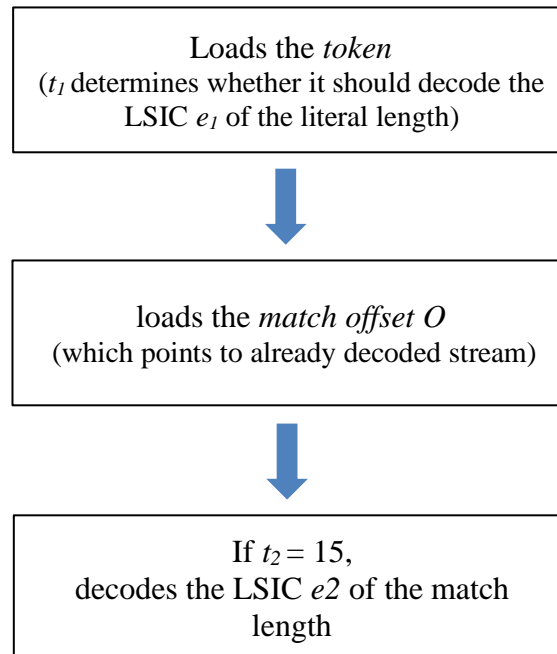


Figure 3.6. Linear Small Integer Code (LSIC) flowchart

The process using LSIC allows the LZ4 provide minimum number of byte used in the case of a short match. The literal L determines the length of the block and the first (highest) field in the token is used to define the literal. Then there is the *match offset* O which is taken from LZ77 family and the *match length* LISC value e_2 which is used with lower part of the token t_2 . In literal, if the field is in maximum value (15), the algorithm will read an integer with LSIC and add it to the original value (15) to obtain the literal lengths. The process of decoding algorithm:





Match offset will copy bytes from already decoded stream to the front of it.

Figure 3.7. Step of LZ4 compression

The compression process will be carried out in the section of *match offset*. It will reduces the size of the input data. This process is also called deduplication (shown in Figure 3.8). The decompression is fast and simple, provides a very short decompression time compared to other algorithms.



Figure 3.8. Deduplication of LZ4

For example, consider the table of some characters shown in table 6.

Character	Code
ab	0
abc	1
baba	2
aa	3

Table 6. LZ4 character information

And we have input data shown in the table 7.

Input Data											
a	b	a	b	c	b	a	b	a	b	a	a

Table 7. LZ4 Input data

If we search the data of 'ab', we will get a partial match of '0'. And if we continue with the other part of the message, we can obtain the value of '1' for the data of 'abc'. This process will be carried out again with other parts of the message. In this matter, we can quickly find out how many bytes they have in common as a prefix. In this case, it is 4 bytes. If in the process we found no match, then we need to write the encoded message as literal until the good match is obtained.

4. Implementation

4.1 Implementation Results

This section will explain the result of the implementation in the experiment. The performance of a lossless compression algorithm usually will be judged by the saving percentage of the compression algorithm (compression ratio), the complexity of the technique, and the usability that refers to the ability of an algorithm to obtain good performances on the different data type. The results are tabulated and analyzed to reach the best technique, advantages, and disadvantages for each one, and when each one is best to use.

The experiments involve four algorithms, Huffman Code, Arithmetic Coding, LZ77, and LZ4. The result will be the value of a percentage compression ratio. In our test, the compression ratio defines as:

$$\text{Compression ratio} = (1 - \frac{\text{size after compression}}{\text{size before compression}}) \times 100\% \quad (1)$$

The equation above is used because we mainly consider spacing saving in this thesis instead of the other popular definition used in many database compression methods, which is to calculate the ratio between the original data size and the compressed size. The reason for adopting this definition is because we want to obtain the objective of the thesis and it is easier to see the absolute size reduction through the space saved. The result will be one of the determining factors in choosing which algorithm is best to be applied to time series data in AWS.

To ensure the validity of the result, the experiments were carried out not only for the data compression part but also for the decompression process. All data that has been compressed must be able to be returned perfectly. The reliability of the experiment can be guaranteed by applying different data types as input and ensure all work perfectly. Measuring the execution time is indicative, but it might be misleading, as it depends on the optimization of the implementation. So at this experiment, we do not consider the time execution as a performance factor.

There are six datasets from six different sources that will be used in the simulation, Table 8 shows each size of the input data.

File Name	Complete Data (KB)	Temperature Data (KB)	Humidity Data (KB)	Wind Speed Data (KB)	Wind Direction Data (KB)
Eixample	4.211	429	164	456	434
Sant Pol de Mar	6.989	480	383	400	372
Hospitalet	1.759	59	60	59	44
Raval	6.144	495	507	457	486
Sarria	6.541	551	556	526	486
Zona Universitaria	6.138	490	504	453	485

Table 8. File size of input data

4.1.1 Huffman Code

The first experiment is a simulation using Huffman Code algorithm. Two input data is given, those are original data and process data. Original data is the data obtained directly from AWS, and process data is the data that has been processed using the computational difference method.

Two types of data are also given, namely complete data and specific data. Complete data is the data that contains all information from the sensor installed in AWS. The specific data is one type of data that is simulated at a time. It consists of data of temperature, humidity, wind speed, and wind direction. All data is taken from six different AWS sources. The name of the simulated file is adjusted to the location where AWS is located.

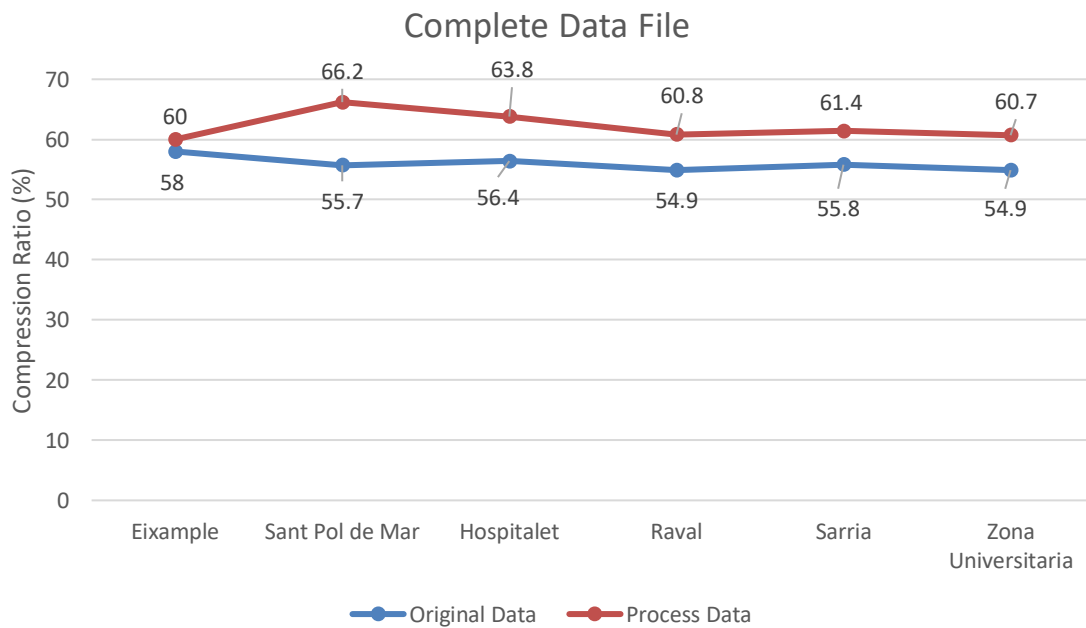


Figure 4.1. Compression ratio of complete data (Huffman Code)

The first graph, which we show in Figure 4.1, plots the percentage of the compression ratio of complete data file after applying the Huffman code algorithm. The blue line shows the performance of original data file, where the highest compression ratio reaches the number of 58%. For process data, we can obtained a higher compression ratio, with the lowest value of process data is 60.7% and the highest compression ratio reaches the value of 66.2%. In general, the value of compression ratio for process data is greater than the original data. This happens because by applying the computational difference method, we can reduce the entropy of the data. In this case, the size of the data will also become smaller.

For all the three data from the government of Barcelona (Raval, Sarria, and Zona Universitaria), the result of compression ratio is almost the same. That is because the amount of data in the three files is relatively the same. It has also the same weather information and the same time span, the difference is the value provided by each AWS. The result obtained in this simulation tends to be stable, with a range between 54.4% and 66.2%.

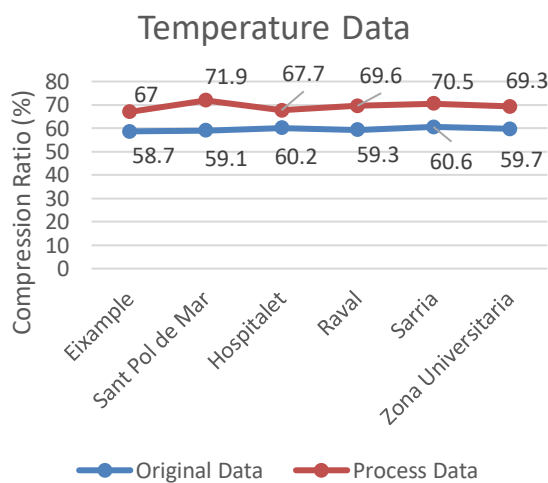


Figure 4.2. Temperature data (Huffman)

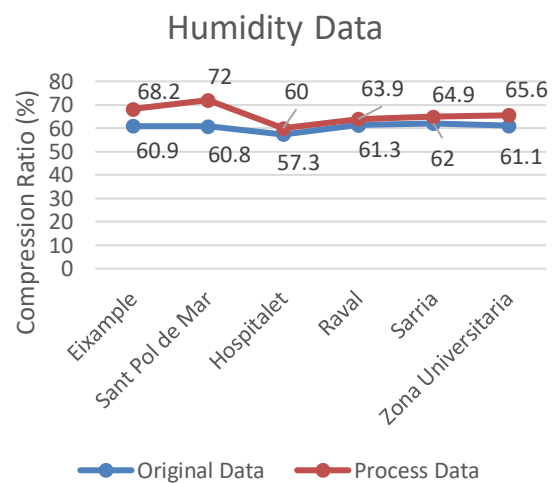


Figure 4.3. Humidity data (Huffman)

The result of compression ratio for a specific data type gives us the same pattern as simulation before, where all the value of process data is greater than the original data. For temperature data, the best compression ratio is at the point of 71.9% using the data process as input, and at 60.6% using original data as input.

Figure 4.3 shows the performance of humidity data. In this simulation, we obtained the same performance result compared to temperature data. The compression ratio of Sant Pol de Mar tends to be the highest between all of it, with a value of 72%.

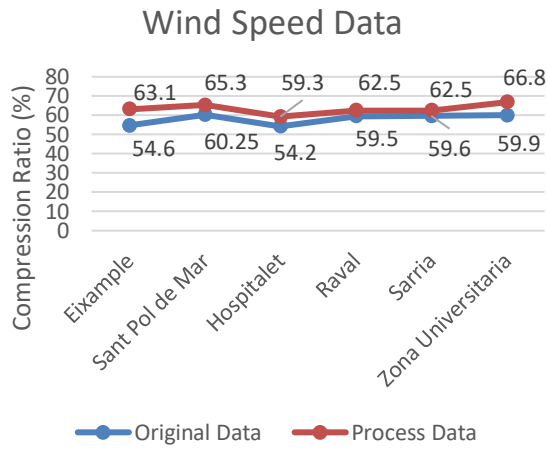


Figure 4.4. Wind Speed data (Huffman)

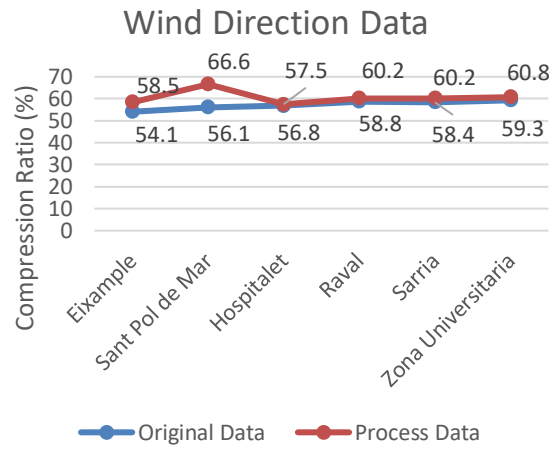


Figure 4.5. Wind Direction data (Huffman)

The next simulation is a simulation of wind speed data and wind direction data. The compression ratio of the data process once again outperforms the original data. The range obtained from process data is 63.1% to 66.8% and 54.1% to 59.3% for original data.

From the simulation of wind direction, an interesting phenomenon has occurred. Where the difference in data compression between the process data and the original data is not so huge for Raval, Sarria, and Zona Universitaria. In fact, this behavior is reasonable since the difference between the original data and data that has been processed by the computational difference method is not so different.

From the five figures, we can conclude that by applying this algorithm, the compression ratio values of all data are mostly in the same range. Both for process data and original data. In our case, the range is between 54.2% to 72%. But the compression ratio cannot reach more than 72% even though we have applied the computational difference method and simulated it with specific data. And the compression ratio result of process data always outperforms the original data.

4.1.2 Arithmetic Coding

The second simulation is applying Arithmetic Coding algorithm to the data. The experiment condition and data used in this simulation are the same as the experiments of Huffman code. Figure 4.6 shows the performance of Arithmetic Coding. As expected, Arithmetic coding yielded slightly better compression ratios than the Huffman coding in terms of compression data. With the same data, Arithmetic Coding can provide data compression of 70%, where the highest compression of Huffman coding is 66.2%.

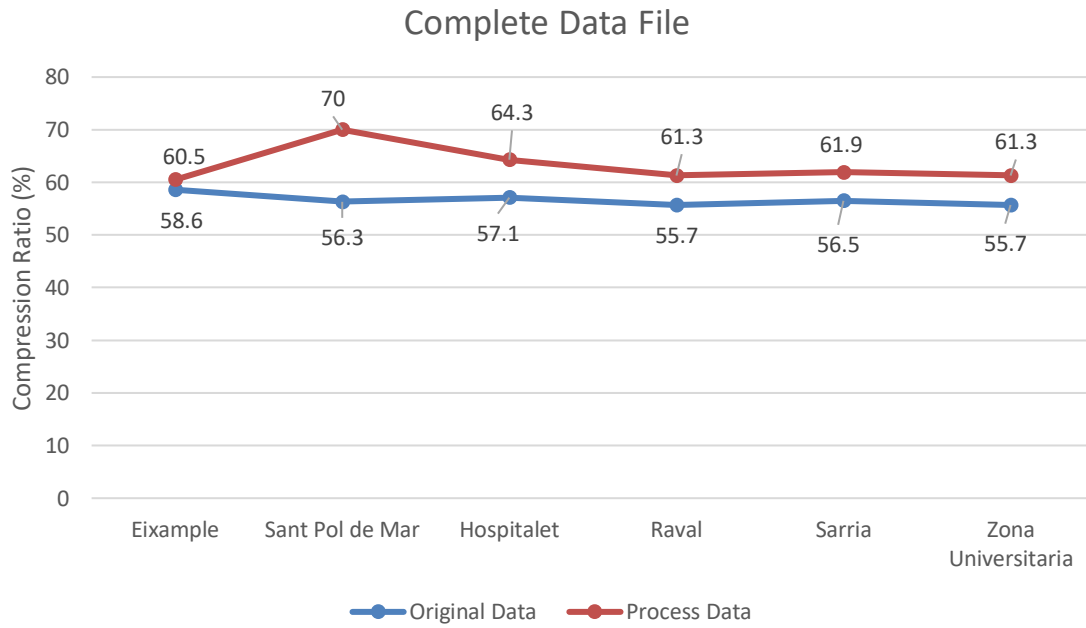


Figure 4.6. Compression ratio of complete data (Arithmetic Coding)

The line red as the representation of process data shows a better performance than the blue line, which is the representation of original data. Stable patterns are obtained from all five data except data from Sant Pol de Mar. The algorithm can compress data from process data of Sant Pol de Mar way better than using original data as input. It is because the time difference values are very similar to each other. In this simulation, we computed the redundancy of each file. We can obtain the fact that the file from Sant Pol de Mar has the greatest redundancy. For this reason, it can provide the highest compression ratio results. The equation to calculate data redundancy show in equation 8. Where C_r is compression ratio.

$$\text{Data Redundancy, RD} = 1 - \frac{1}{C_r} \quad (8)$$

File Name	Redundancy (Original Data)	Redundancy (Compress Data)
Eixample	0.982	0.983
Sant Pol de Mar	0.982	0.985
Hospitalet	0.982	0.984
Raval	0.982	0.983
Sarria	0.982	0.983
Zona Universitaria	0.982	0.983

Table 9. Redundancy of simulation (Arithmetic)

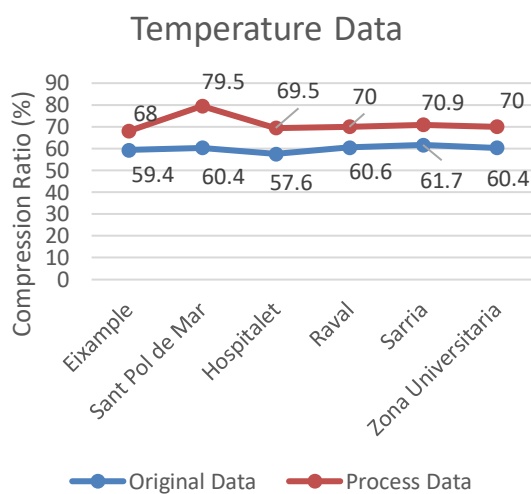


Figure 4.7. Temperature data (Arithmetic)

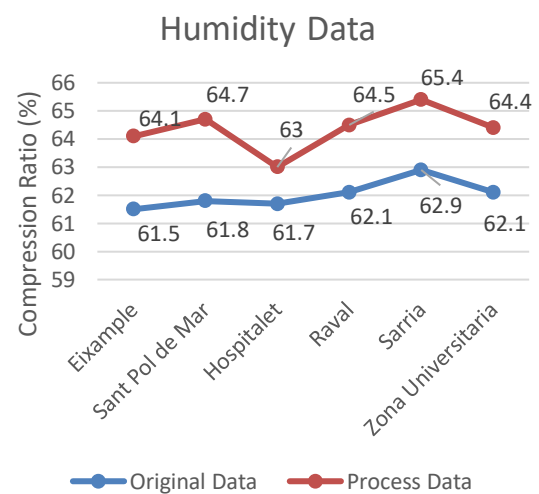


Figure 4.8. Humidity data (Arithmetic)

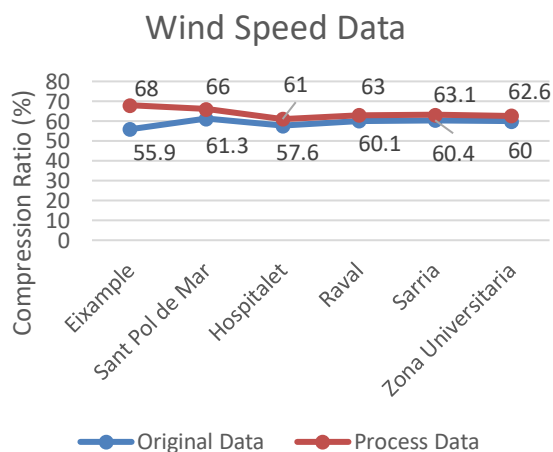


Figure 4.9. Wind Speed data (Arithmetic)

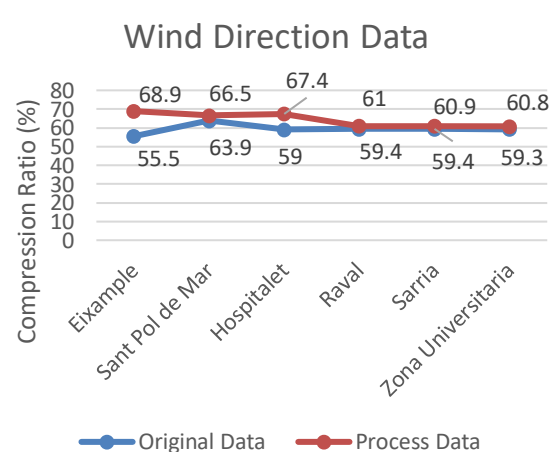


Figure 4.10. Wind Direction data (Arithmetic)

When we applied specific data for the input, all process data outperform the results of the original data. On the wind direction simulation, the difference of process data and original data

is not very visible. But in the simulation of temperature, the difference even reached 19.5%. That is because of the difference in the entropy values of each data, where the greater the entropy, the smaller the compression ratio. From the input file, two-digit numbers of data is presented in the temperature data, and three-digit numbers of data are presented in wind direction data. The Arithmetic coding is marginally better than Huffman code, mainly in the application of time series data file.

4.1.3 LZ77

The next simulation is using LZ77 algorithm. The LZ77 Compression Algorithm is used to analyze input data and determine how to reduce the size of that input data by replacing redundant information with metadata. Every duplicate occurrence of data is replaced with the pointer to its first occurrence. The input data and the experimental environment are arranged the same. From figure 4.11 the highest compression ratio is obtained in the simulation using data of Eixample. This can be understood because in Eixample file, some data from one column to another is the same. This behavior also confirms the theorem of LZ77 algorithm that the result of compressing data is higher if the processed data contains data that has been stored in the dictionary.

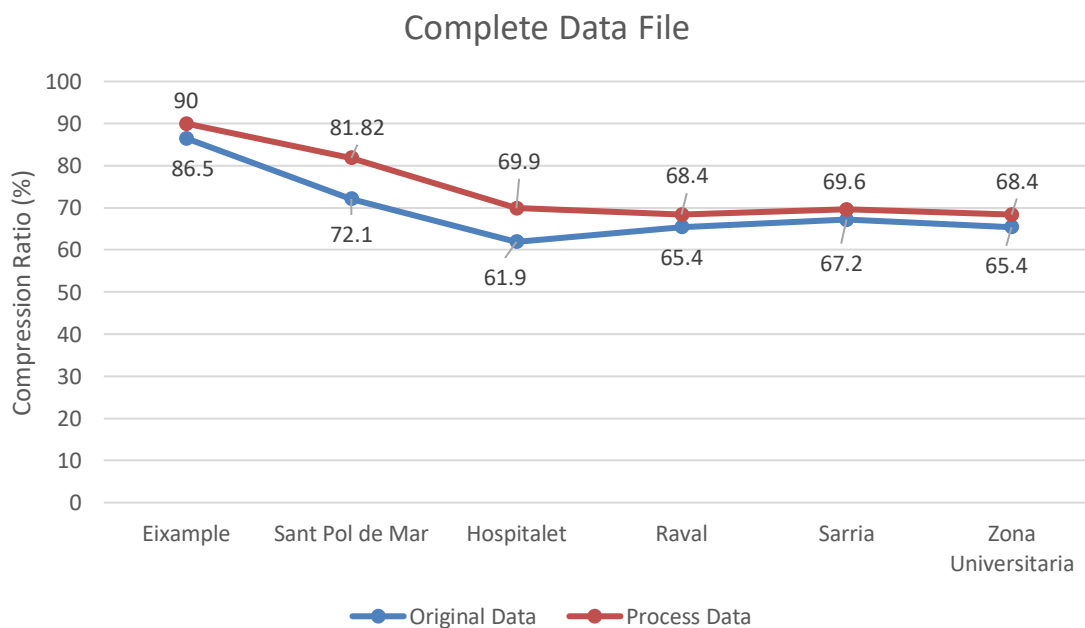


Figure 4.11. Compression ratio of complete data (LZ77)

With complete data as an input, the highest compression ratio can be reached up to 90%. And the lowest compression ratio is 61.9%, it is better than using both Huffman code and Arithmetic code. The average compression ratio is 74.6% for process data and 69.75% for original data. In fact, the result of process data still outperforms the results of the original data.

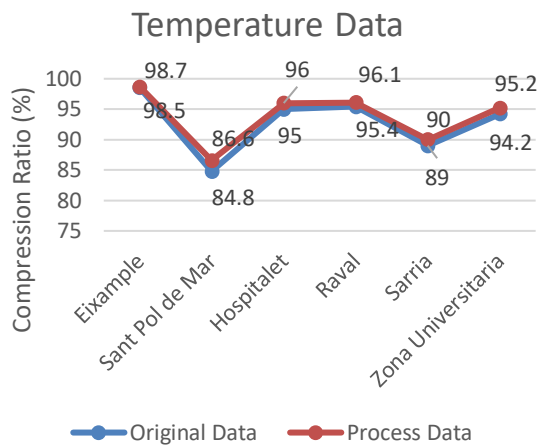


Figure 4.12. Temperature data (LZ77)

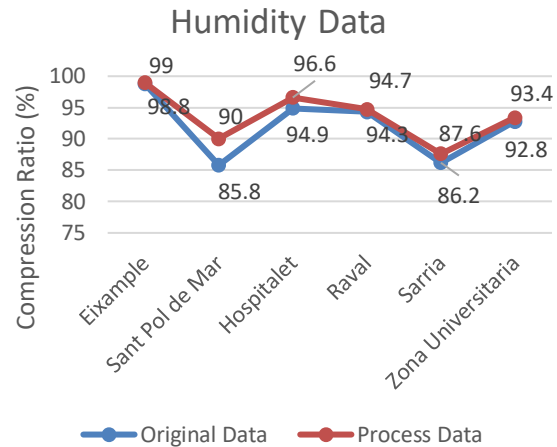


Figure 4.13. Humidity data (LZ77)

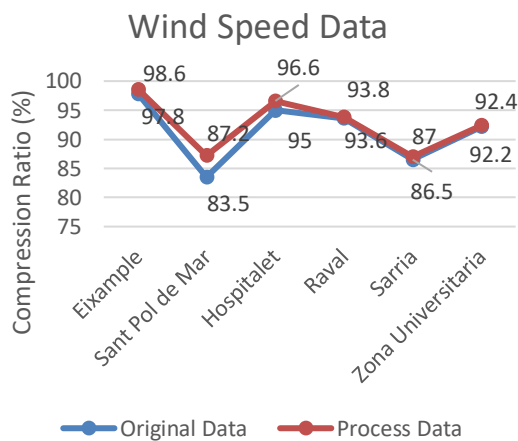


Figure 4.14. Wind Speed data (LZ77)

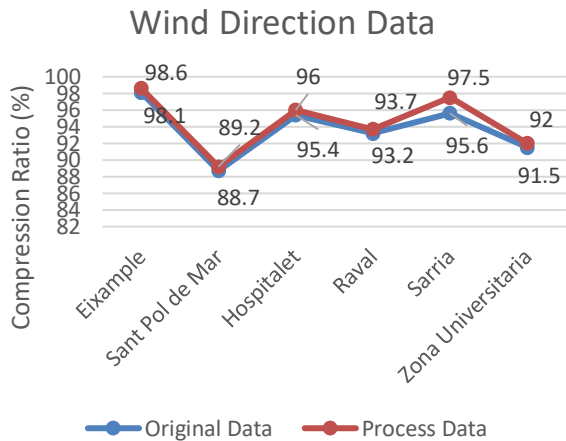


Figure 4.15. Wind Direction data (LZ77)

We have also run the specific data type simulation presented in figure 4.14, figure 4.15, figure 4.16, and figure 4.17, there is a significant difference from the pattern of the graph that was previously obtained. Where at this simulation, we can see the wave pattern of the line. We obtained a drop point in file of Sant Pol de Mar, it is indicated that the data in the file has many different values inside, so it requires more dictionary information. LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that

data existing earlier in the uncompressed data stream. The compression ratio of Eixample always gives the best value among all.

An interesting fact revealed from the above results is that by using specific data we get higher data compression compared to the result of complete data. The lowest compression ratio of specific data is 83.5% for the original data, it is 21.6% higher than the result from the complete data. And 9% improvement for process data.

4.1.4 LZ4

The last experiment is simulation using LZ4 algorithm. The LZ4 algorithm represents the data as a series of sequences. Higher compression ratios can be achieved by investing more effort in finding the best matches of information in the file. The input data and the experimental environment are arranged the same.

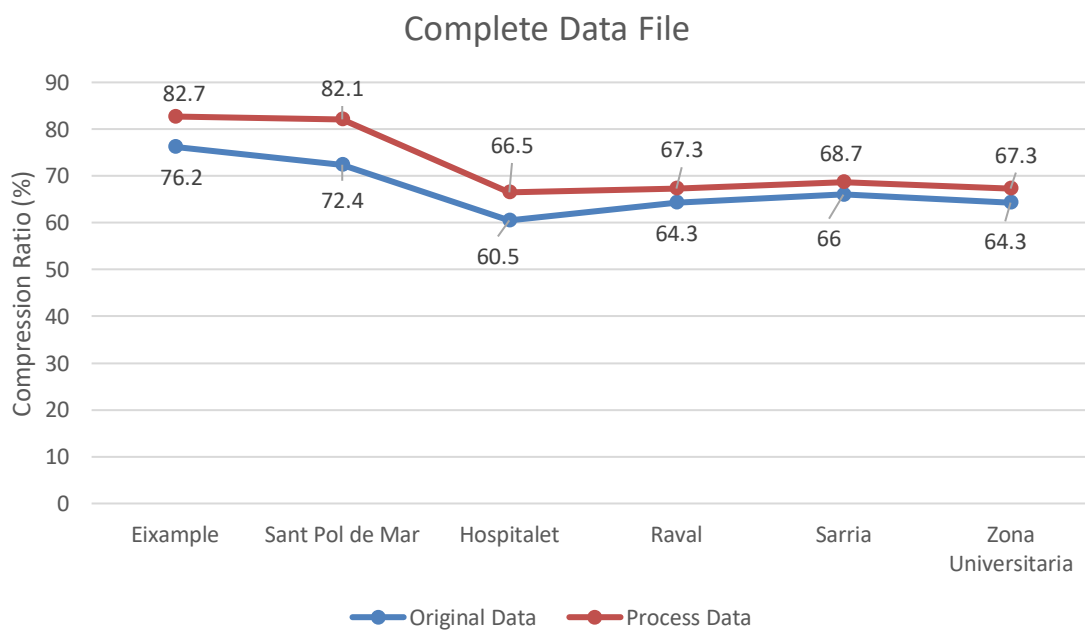


Figure 4.16. Compression ratio of complete data (LZ4)

The overall appearance of the plot, we observe that the compression ratio of the complete data file reached 82.7% and 82.1% for Eixample and Sant Pol de Mar. The average of 67% compression ratio shown in the result of Hospitalet, Raval, Sarria, and Zona Universitaria. In all four files, there are not many changes and the results located at almost the same point in the compression ratio.

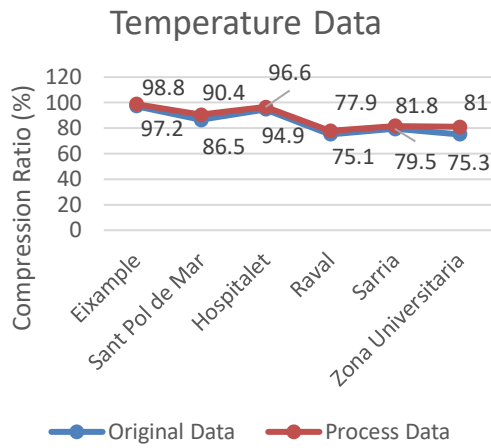


Figure 4.17. Temperature data (LZ4)

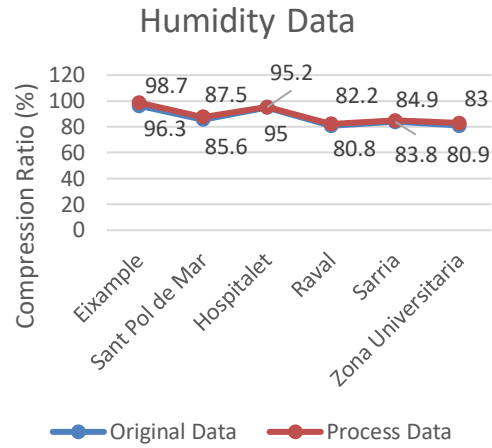


Figure 4.18. Humidity data (LZ4)

Figure 4.17 shows the result of temperature data using LZ4 algorithm. 98.8% of the compression ratio can be obtained in this simulation. If we observe, the pattern on the graph looks more stable compared to the results obtained in experiments using the LZ77 algorithm. Figure 4.18 shows the result of humidity data. Where we obtained a compression ratio average of 88.5% for original data and 87% for process data.

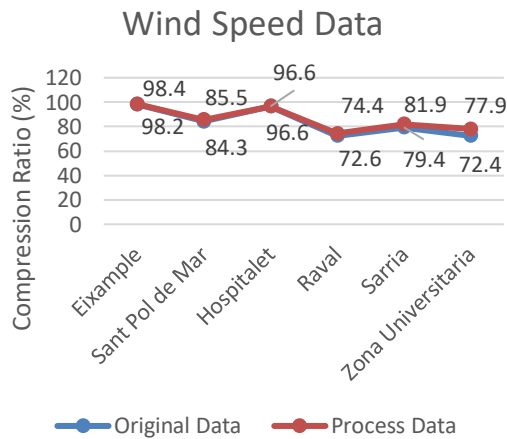


Figure 4.19. Wind Speed data (LZ4)

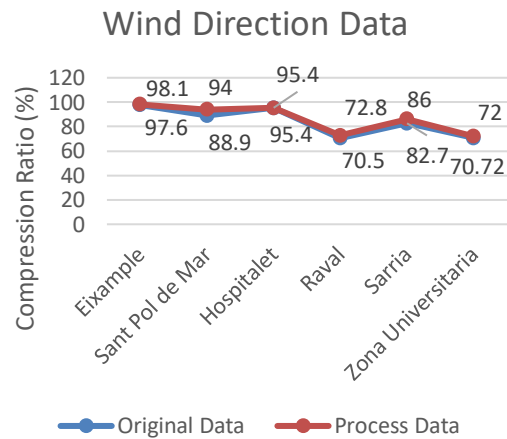


Figure 4.20. Wind Direction data (LZ4)

Similar plots for the wind speed data and wind direction data are depicted in Figure 4.19 and Figure 4.20. The compression ratio average from wind speed is 83.9% for original data, and 85.7% for process data. Even in the result of Sant Pol de Mar, both original data and process data shows the same compression ratio result. For the wind direction data, the algorithm can compress the file until 98.1% for process data, and 97.6% for original data.

4.1.5 Comparison of 4 Algorithms

Finally, we compared all the algorithms in this thesis, Huffman Code, Arithmetic Coding, LZ77, and LZ4. The result can be found in figure 4.21 until Figure 4.25. To obtained the chart, we took the average number of compression ratios from each simulation, both for original data and process data.

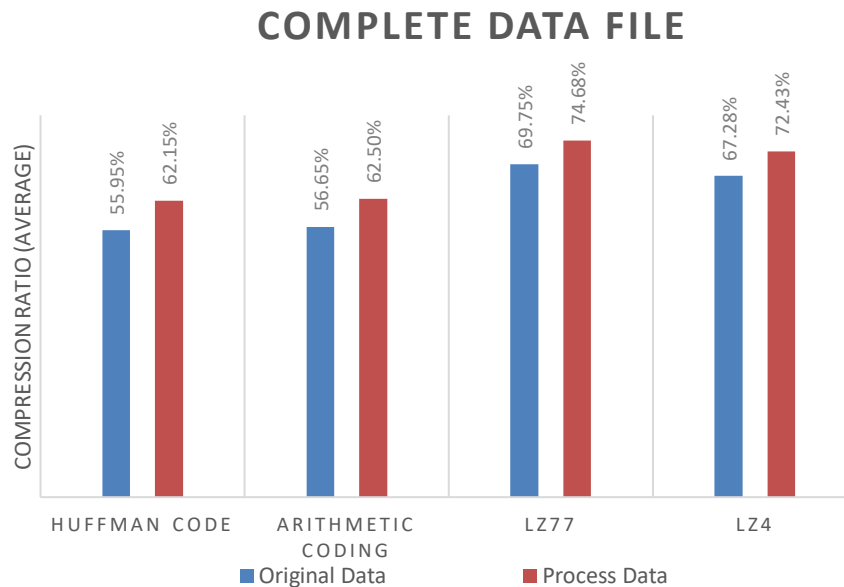


Figure 4.21. Compression ratio average of complete data

The first chart shows the average value of the compression ratio of the complete data file. Process data applied in all algorithms outperform the original data. Observe that we obtained the highest compression ratio using LZ77 algorithm with a value of 74.68%. The performance improvement varies from 55.95% to as much as 74.68%. The performance of Huffman code and Arithmetic Coding slightly the same, although arithmetic is still superior with a difference of only less than 1% of data compression.

The figure below shows the simulation results using specific data.

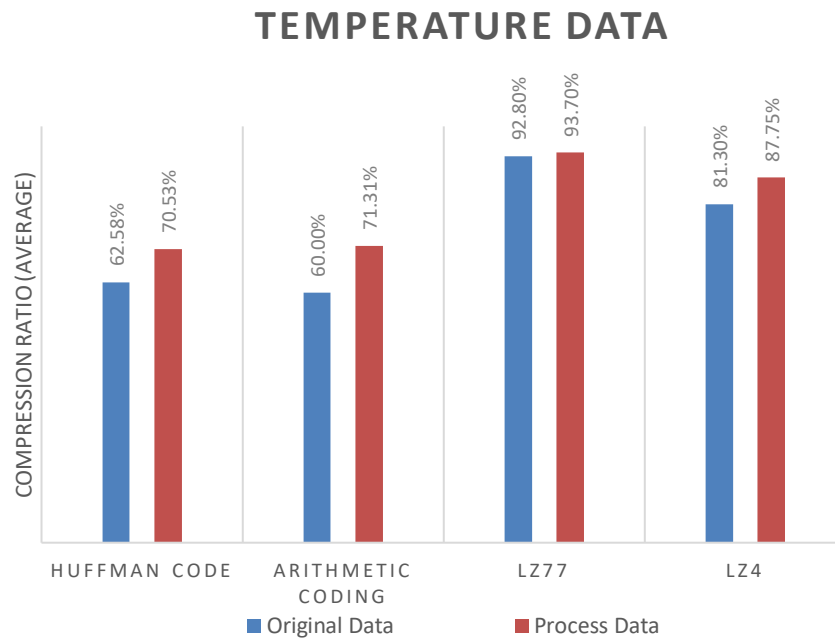


Figure 4.22. Compression ratio average of temperature data

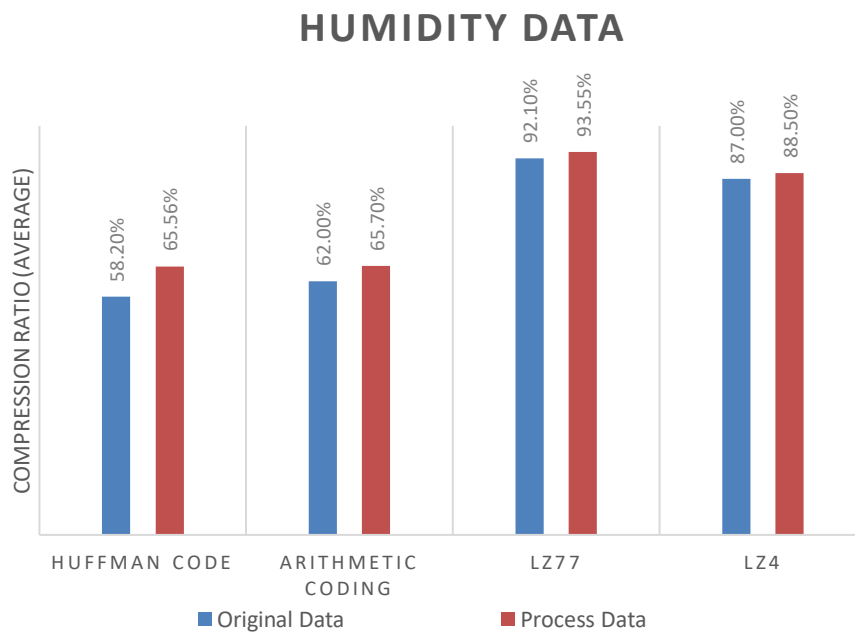


Figure 4.23. Compression ratio average of humidity data

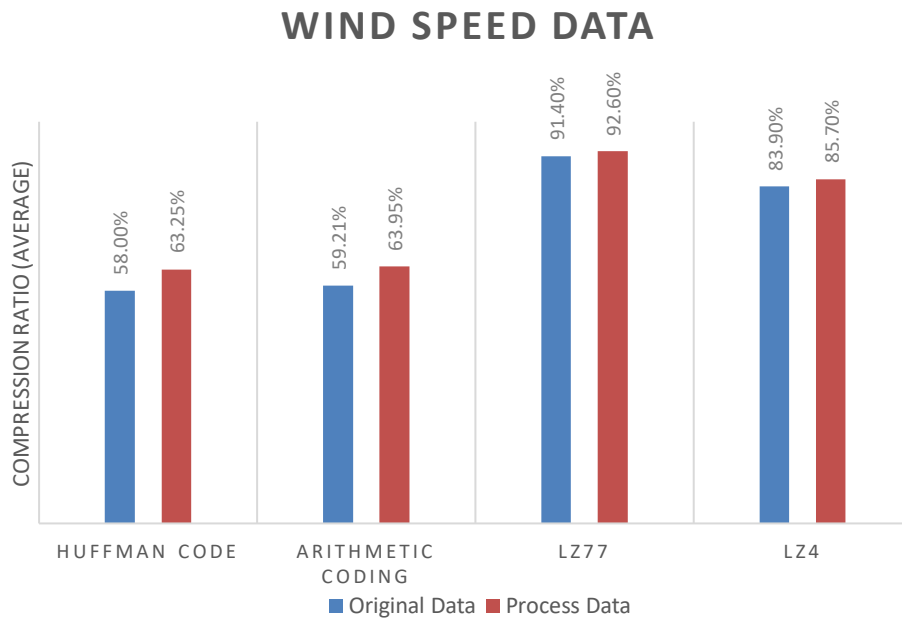


Figure 4.24. Compression ratio average of wind speed data

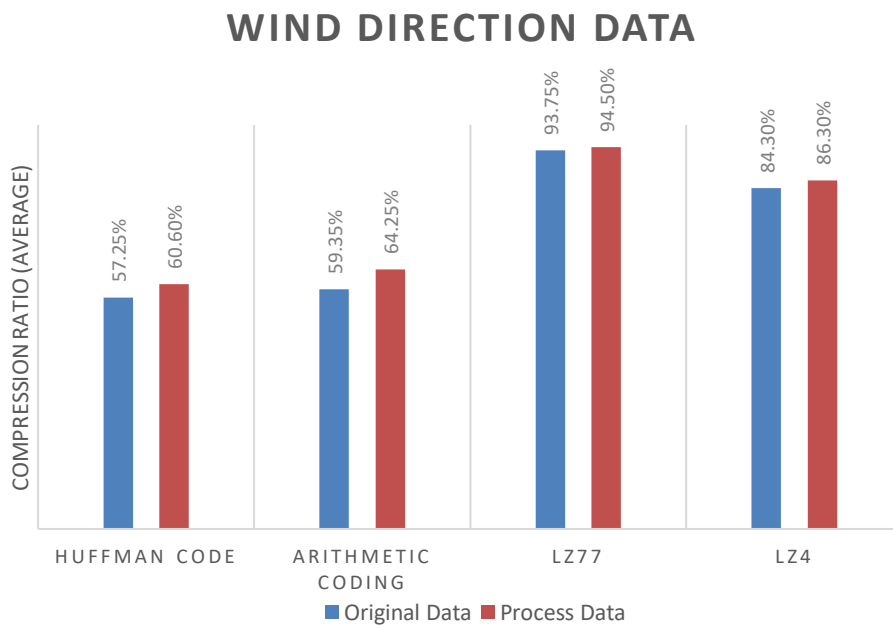


Figure 4.25. Compression ratio average of wind direction data

First, as expected, Arithmetic Coding can give us a better result than the Huffman Code. The results range from 57.25% to 62.58% for the original data, and 60.6% to 71.31% for process data. Another interesting aspect is that LZ77 and LZ4 work very well when we used only one type of data in the simulation. This can be seen from the results of the compression ratio which reached more than 85% in all specific data simulations. This is due to the workings of the algorithm that can compress the data better by finding sequences of data that are repeated.

Moreover, LZ77 is extremely competitive with LZ4, since it dominates it in all of the data compression results. The compression ratio of the process data in all simulation still shows higher results than the results of the original data. Consequently, by applying the computational method, the results of the simulation is improved.

5. Conclusion and Future Work

5.1 Conclusion

The result of these experiment prove that the lossless compression can be beneficially applied on time series data. We created the simulation of Huffman Code, Arithmetic Coding, LZ77, and LZ4. We have explained these algorithms in detail, and tested using different type of file and size. The main conclusions of this thesis are the following:

- From the complexity technique, Arithmetic Coding is more complicated than the Huffman Code. This algorithm processes data as a block of symbols instead of a single symbol. It makes Arithmetic can yield more compression ratio shows by the result of the simulation compared to the Huffman code.
- LZ77 and LZ4 work very well when the data is repeated. It performs almost equally and are the most effective of all the algorithms we tested. In the specific data case, where the data contains repeated values, this algorithm can compress data up to 99.8%. LZ77 has greater technical complexity than other algorithms, so it requires a more sophisticated processor. Even so, this algorithm is still possible to be applied to AWS, which generally has low processing capability and low energy supply.
- For the time series data of AWS, it can be concluded that LZ77 provides the best result than the other three methods. The values are drawn from the four algorithms based on compressed file size. It can be considered as the most efficient algorithm among the selected ones to be applied in AWS.
- Process data in all simulations show a better result than the original data. It helps to perform a lossless compression achieving higher compression ratio because with applying the computational difference method, we can reduce the entropy of the data.
- Specific data file shows better compression ratio results in all algorithms than the complete data file. That is because, with the more diverse types of data in a file, the compression process will be more complicated and the entropy becomes greater. It will produce a low compression ratio.

5.2 Future Work

This section describes some suggestions for future work that might follow the work described in this thesis.

- In this thesis we implemented four algorithms. Henceforth, other algorithms can be tested. To find out the effect of the other algorithm on time series data from AWS.
- Calculate the communication cost reduction from data transmission based on the compression ratio that has been obtained. So that we can find out the benefits gained from an economic perspective and can be an opportunity to be applied in real terms.
- Investigate the energy usage of the algorithm due to the popularity of green power. Ideally, with the data compression process, we can obtain energy savings.
- Use multiple data time series as input and compress at the same time. Because in this experiment we simulate one data at a time.

6. Bibliography

- [1] Perwej, Y., Kerim, B., Ahmed, M., and Harb, H., “An Extended Review on Internet of Things (IoT) and its Promising Applications”, Al Baha University, Al Baha Kingdom of Saudi Arabia, 2019.
- [2] Cheng, L., Leung, A., and Ozawa, S., “Neural Information Processing”, Springer Nature, Switzerland, 2018.
- [3] Huffman, D., “A Method for the Construction of Minimum-Redundancy Codes”, Massachusetts Institute of Technology, Cambridge, 1952.
- [4] Salomon, D., Motta, G., “Handbook of Data Compression Fifth edition”, Springer, 2010.
- [5] Chen, H. C. and Wang, Y. L. and Lan, Y. F., “A Memory Efficient and Fast Huffman Decoding Algorithm”, Information Processing Letters, Vol. 69, No. 3, pp. 119- 122, 1999.
- [6] Buro. M., “On the maximum length of Huffman codes”, Information Processing Letters, Vol. 45, No.5, pp. 219-223, April 1993.
- [7] Gallager, R.. “Variations on a theme by Huffman”, IEEE Transactions on Information Theory, 24(6):668–674, 1978.
- [8] Engineering and Technology History Wiki, “History of Lossless Data Compression Algorithms”, ETHW website. 15-Jun-2018 [Online]. Available: http://ethw.org/History_of_Lossless_Data_Compression_Algorithms. [Accessed: 4-June-2020].
- [9] Bartik, M. and Ubik, S., “LZ4 compression algorithm on FPGA”, IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Dec 2015.
- [10] Bjärås, F., “Comparative study of compression algorithms on time Series Data for IoT Devices”, Lund University, July 2019.
- [11] Xing, R., “The Compression of IoT Operational Data Time Series in Vehicle Embedded Systems”, KTH Royal Institute of Technology, Sweden, 2018.
- [12] Zaitsev, P. and Tkachenko, V., “Evaluating Database Compression Methods: Update”, Percona Database Performance Blog. 13-Apr-2016 [Online]. Available: <https://www.percona.com/blog/2016/04/13/evaluating-database-compressionmethods-update/>. [Accessed: 23-Apr-2018]
- [13] Deepu, C., Heng, C., and Lian, Y., “A Hybrid Data Compression Scheme for Power Reduction in Wireless Sensors for IoT”, IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS, VOL. 11, NO. 2, April, 2017.

- [14] Pope, J., Vafeas, A., Elsts, A., Oikonomou, G., Pienchoki, R., and Craddock, I., “An Accelerometer Lossless Compression Algorithm and Energy Analysis for IoT Devices”, Wireless Communications and Networking Conference Workshops (WCNCW), IEEE, April, 2018.
- [15] Vadrevu, S., and Manikandan, M., “A New Quality-Aware Quality-Control Data Compression Framework for Power Reduction in IoT and Smartphone PPG Monitoring Devices”, IEEE Sensors Letters, Volume 3, July 2019.
- [16] Witten, I., Neal, R., and Cleary, J., “Arithmetic coding for data compression,” Commun. ACM, vol. 30(6), pp. 520–540, June 1987.
- [17] Moon, A. Y., Kim, J., Zhang, J., and Son, S., “Lossy Compression on IoT Big Data by Exploiting Spatiotemporal Correlation”, High Performance Extreme Computing Conference (HPEC), IEEE, September, 2017.
- [18] Park, J., Park, H., and Choi, Y., “Data Compression and Prediction Using Machine Learning for Industrial IoT”, International Conference on Information Networking (ICOIN), IEEE, January, 2018.
- [19] Chatterjee, A., Shah, R., and Hasan, K., “Efficient Data Compression for IoT Devices using Huffman Coding Based Techniques”, International Conference on Big Data (Big Data), IEEE, December, 2018.
- [20] Sarbishei, O., “Refined Lightweight Temporal Compression for Energy-Efficient Sensor Data Streaming”, 5th World Forum on Internet of Things (WF-IoT), IEEE, April, 2019.
- [21] Ez-Zazi, I., Arioua, M., and Oualkadi, A., “Analysis of Lossy Compression and Channel Coding Tradeoff For Energy Efficient Transmission in Low Power Communication Systems”, 9th International Symposium on Signal, Image, Video and Communications (ISIVC), IEEE, November, 2018.
- [22] Williams, R., “Adaptive Data Compression”, Springer, 1st edition, 1991.

7. Appendix

A. Table of Simulation Result

1. Huffman Code: Complete Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	1,766 KB	58 %
Sant Pol de Mar	6,989 KB	3,090 KB	55.7 %
Hospitalet	1,759 KB	766 KB	56.45 %
Raval	6,144 KB	2766 KB	54.9 %
Sarria	6,541 KB	2888 KB	55.8 %
Z. Universitaria	6,138 KB	2763 KB	54.9 %

2. Huffman Code: Complete Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	1,686 KB	60 %
Sant Pol de Mar	6,989 KB	2,251 KB	66.2 %
Hospitalet	1,759 KB	636 KB	63.8 %
Raval	6,144 KB	2,407 KB	60.8 %
Sarria	6,541 KB	2,521 KB	61.4 %
Z. Universitaria	6,138 KB	2,405 KB	60.7 %

3. Huffman Code: Temperature Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	177 KB	58.7 %
Sant Pol de Mar	480 KB	196 KB	59.1 %
Hospitalet	59 KB	26 KB	60.2 %
Raval	495 KB	201 KB	59.3 %
Sarria	551 KB	217 KB	60.6 %
Z. Universitaria	490 KB	197 KB	59.7 %

4. Huffman Code: Temperature Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	140 KB	67 %
Sant Pol de Mar	480 KB	100 KB	72 %
Hospitalet	59 KB	19 KB	60 %
Raval	495 KB	150 KB	63.9 %
Sarria	551 KB	162 KB	64.9 %
Z. Universitaria	490 KB	150 KB	65.6 %

5. Huffman Code: Humidity Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	64 KB	60.9 %
Sant Pol de Mar	383 KB	150 KB	60.8 %
Hospitalet	60 KB	26 KB	57.3 %
Raval	507 KB	196 KB	61.3 %
Sarria	556 KB	211 KB	62 %
Z. Universitaria	504 KB	196 KB	61.1 %

6. Huffman Code: Humidity Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	52 KB	68.2 %
Sant Pol de Mar	383 KB	107 KB	72 %
Hospitalet	60 KB	24 KB	60 %
Raval	507 KB	183 KB	63.9 %
Sarria	556 KB	195 KB	64.9 %
Z. Universitaria	504 KB	183 KB	65.6 %

7. Huffman Code: Wind Speed Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	207 KB	54.6 %
Sant Pol de Mar	400 KB	159 KB	60.25 %
Hospitalet	59 KB	27 KB	54.2 %
Raval	457 KB	185 KB	59.5 %
Sarria	526 KB	212 KB	59.6 %
Z. Universitaria	457 KB	183 KB	59.9 %

8. Huffman Code: Wind Speed Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	168 KB	63.1 %
Sant Pol de Mar	400 KB	139 KB	65.3 %
Hospitalet	59 KB	24 KB	59.3 %
Raval	457 KB	171 KB	62.5 %
Sarria	526 KB	197 KB	62.5 %
Z. Universitaria	457 KB	150 KB	66.8 %

9. Huffman Code: Wind Direction Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	199 KB	54.1 %
Sant Pol de Mar	372 KB	136 KB	56.1 %
Hospitalet	44 KB	19 KB	56.8 %
Raval	486 KB	200 KB	58.8 %
Sarria	486 KB	202 KB	58.4 %
Z. Universitaria	485 KB	197 KB	59.3 %

10. Huffman Code: Wind Direction Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	180 KB	58.5 %
Sant Pol de Mar	372 KB	124 KB	66.6 %
Hospitalet	44 KB	20 KB	57.5 %
Raval	486 KB	193 KB	60.2 %
Sarria	486 KB	194 KB	60.2 %
Z. Universitaria	485 KB	190 KB	60.8 %

11. Arithmetic Coding: Complete Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	1,740 KB	58.6 %
Sant Pol de Mar	6,989 KB	3,050 KB	56.3 %
Hospitalet	1,759 KB	754 KB	57.1 %
Raval	6,144 KB	2,718 KB	55.7 %
Sarria	6,541 KB	2,841 KB	56.5 %
Z. Universitaria	6,138 KB	2,715 KB	55.7 %

12. Arithmetic Coding: Complete Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	1,662 KB	60.5 %
Sant Pol de Mar	6,989 KB	2,232 KB	70 %
Hospitalet	1,759 KB	627 KB	64.3 %
Raval	6,144 KB	2,374 KB	61.3 %
Sarria	6,541 KB	2,491 KB	61.9 %
Z. Universitaria	6,138 KB	2,372 KB	61.3 %

13. Arithmetic Coding: Temperature Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	174 KB	59.4 %
Sant Pol de Mar	480 KB	190 KB	60.4 %
Hospitalet	59 KB	25 KB	57.6 %
Raval	495 KB	195 KB	60.6 %
Sarria	551 KB	211 KB	61.7 %
Z. Universitaria	490 KB	194 KB	60.4 %

14. Arithmetic Coding: Temperature Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	137 KB	68 %
Sant Pol de Mar	480 KB	97 KB	79.5 %
Hospitalet	59 KB	18 KB	69.5 %
Raval	495 KB	148 KB	70 %
Sarria	551 KB	160 KB	70.9 %
Z. Universitaria	490 KB	147 KB	70 %

15. Arithmetic Coding: Humidity Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	63 KB	61.5 %
Sant Pol de Mar	383 KB	146 KB	61.8 %
Hospitalet	60 KB	25 KB	61.7 %
Raval	507 KB	192 KB	62.1 %
Sarria	556 KB	206 KB	62.9 %
Z. Universitaria	504 KB	191 KB	62.1 %

16. Arithmetic Coding: Humidity Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	51 KB	68.9 %
Sant Pol de Mar	383 KB	105 KB	72.5 %
Hospitalet	60 KB	23 KB	61.6 %
Raval	507 KB	180 KB	64.5 %
Sarria	556 KB	192 KB	65.4 %
Z. Universitaria	504 KB	77 KB	64.4 %

17. Arithmetic Coding: Wind Speed Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	201 KB	55.9 %
Sant Pol de Mar	400 KB	155 KB	61.3 %
Hospitalet	59 KB	25 KB	57.6 %
Raval	457 KB	182 KB	60.1 %
Sarria	526 KB	208 KB	60.4 %
Z. Universitaria	457 KB	181 KB	60 %

18. Arithmetic Coding: Wind Speed Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	164 KB	64 %
Sant Pol de Mar	400 KB	136 KB	66 %
Hospitalet	59 KB	23 KB	61 %
Raval	457 KB	169 KB	63 %
Sarria	526 KB	194 KB	63.1 %
Z. Universitaria	457 KB	169 KB	62.6 %

19. Arithmetic Coding: Wind Direction Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	193 KB	55.5 %
Sant Pol de Mar	372 KB	134 KB	63.9 %
Hospitalet	44 KB	18 KB	59 %
Raval	486 KB	197 KB	59.4 %
Sarria	486 KB	197 KB	59.4 %
Z. Universitaria	485 KB	198 KB	59.3 %

20. Arithmetic Coding: Wind Direction Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	176 KB	59.4 %
Sant Pol de Mar	372 KB	121 KB	67.4 %
Hospitalet	44 KB	19 KB	56.8 %
Raval	486 KB	191 KB	58.1 %
Sarria	486 KB	190 KB	60.9 %
Z. Universitaria	485 KB	191 KB	58 %

21. LZ77: Complete Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	568 KB	86.5 %
Sant Pol de Mar	6,989 KB	945 KB	72.1 %
Hospitalet	1,759 KB	669 KB	61.9 %
Raval	6,144 KB	2,125 KB	65.4 %
Sarria	6,541 KB	2,145 KB	67.2 %
Z. Universitaria	6,138 KB	2,122 KB	65.4 %

22. LZ77: Complete Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	420 KB	90 %
Sant Pol de Mar	6,989 KB	1,270 KB	81.82 %
Hospitalet	1,759 KB	529 KB	69.9 %
Raval	6,144 KB	1,939 KB	68.4 %
Sarria	6,541 KB	1,984 KB	69.6 %
Z. Universitaria	6,138 KB	1,937 KB	68.4 %

23. LZ77: Temperature Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	5 KB	98.8 %
Sant Pol de Mar	480 KB	68 KB	85.8 %
Hospitalet	59 KB	3 KB	94.9 %
Raval	495 KB	28 KB	94.3 %
Sarria	551 KB	76 KB	86.2 %
Z. Universitaria	490 KB	35 KB	92.8 %

24. LZ77: Temperature Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	4 KB	99 %
Sant Pol de Mar	480 KB	48 KB	90 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	495 KB	26 KB	94.7 %
Sarria	551 KB	68 KB	87.6 %
Z. Universitaria	490 KB	32 KB	93.4 %

25. LZ77: Humidity Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	2 KB	98.7 %
Sant Pol de Mar	383 KB	58 KB	84.8 %
Hospitalet	60 KB	3 KB	95 %
Raval	507 KB	23 KB	95.4 %
Sarria	556 KB	61 KB	89 %
Z. Universitaria	504 KB	29 KB	94.2 %

26. LZ77: Humidity Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	2 KB	98.7 %
Sant Pol de Mar	383 KB	51 KB	86.6 %
Hospitalet	60 KB	3 KB	95 %
Raval	507 KB	22 KB	95.6 %
Sarria	556 KB	61 KB	89 %
Z. Universitaria	504 KB	28 KB	94.4 %

27. LZ77: Wind Speed Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	10 KB	97.8 %
Sant Pol de Mar	400 KB	66 KB	83.5 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	457 KB	29 KB	93.6 %
Sarria	526 KB	71 KB	86.5 %
Z. Universitaria	457 KB	35 KB	92.2 %

28. LZ77: Wind Speed Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	6 KB	98.6 %
Sant Pol de Mar	400 KB	51 KB	87.25 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	457 KB	28 KB	93.8 %
Sarria	526 KB	68 KB	87 %
Z. Universitaria	457 KB	34 KB	92.4 %

29. LZ77: Wind Direction Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	8 KB	98.1 %
Sant Pol de Mar	372 KB	42 KB	88.7 %
Hospitalet	44 KB	2 KB	95.4 %
Raval	486 KB	33 KB	93.2 %
Sarria	486 KB	21 KB	95.6 %
Z. Universitaria	485 KB	41 KB	91.5 %

30. LZ77: Wind Direction Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	6 KB	98.6 %
Sant Pol de Mar	372 KB	43 KB	88.4 %
Hospitalet	44 KB	2 KB	95.4 %
Raval	486 KB	33 KB	93.2 %
Sarria	486 KB	22 KB	99.5 %
Z. Universitaria	485 KB	42 KB	91.3 %

31. LZ4: Complete Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	1,001 KB	76.2 %
Sant Pol de Mar	6,989 KB	1,927 KB	72.4 %
Hospitalet	1,759 KB	694 KB	60.5 %
Raval	6,144 KB	2,190 KB	64.3 %
Sarria	6,541 KB	2,219 KB	66 %
Z. Universitaria	6,138 KB	2,188 KB	64.3 %

32. LZ4: Complete Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	4,211 KB	730 KB	82.7 %
Sant Pol de Mar	6,989 KB	1,250 KB	82.1 %
Hospitalet	1,759 KB	589 KB	66.5 %
Raval	6,144 KB	2,007 KB	67.3 %
Sarria	6,541 KB	2,046 KB	68.7 %
Z. Universitaria	6,138 KB	2,005 KB	67.3 %

33. LZ4: Temperature Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	12 KB	97.2 %
Sant Pol de Mar	480 KB	65 KB	86.5 %
Hospitalet	59 KB	3 KB	94.9 %
Raval	495 KB	123 KB	75.1 %
Sarria	551 KB	113 KB	79.5 %
Z. Universitaria	490 KB	121 KB	75.3 %

34. LZ4: Temperature Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	429 KB	5 KB	98.8 %
Sant Pol de Mar	480 KB	46 KB	90.4 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	495 KB	109 KB	77.9 %
Sarria	551 KB	100 KB	81.8 %
Z. Universitaria	490 KB	93 KB	81 %

35. LZ4: Humidity Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	6 KB	96.3 %
Sant Pol de Mar	383 KB	55 KB	85.6 %
Hospitalet	60 KB	3 KB	95 %
Raval	507 KB	97 KB	80.8 %
Sarria	556 KB	90 KB	83.8 %
Z. Universitaria	504 KB	96 KB	80.9 %

36. LZ4: Humidity Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	164 KB	2 KB	98.7 %
Sant Pol de Mar	383 KB	48 KB	87.5 %
Hospitalet	60 KB	3 KB	95 %
Raval	507 KB	95 KB	81.2 %
Sarria	556 KB	89 KB	83.9 %
Z. Universitaria	504 KB	95 KB	81.1 %

37. LZ4: Wind Speed Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	8 KB	98.3 %
Sant Pol de Mar	400 KB	63 KB	84.3 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	457 KB	125 KB	72.6 %
Sarria	526 KB	108 KB	79.4 %
Z. Universitaria	457 KB	125 KB	72.4 %

38. LZ4: Wind Speed Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	456 KB	7 KB	98.4 %
Sant Pol de Mar	400 KB	58 KB	85.5 %
Hospitalet	59 KB	2 KB	96.6 %
Raval	457 KB	118 KB	74.1 %
Sarria	526 KB	95 KB	81.9 %
Z. Universitaria	457 KB	100 KB	77.9 %

39. LZ4: Wind Direction Data File (Original Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	10 KB	97.6 %
Sant Pol de Mar	372 KB	41 KB	88.9 %
Hospitalet	44 KB	2 KB	95.4 %
Raval	486 KB	143 KB	70.5 %
Sarria	486 KB	84 KB	82.7 %
Z. Universitaria	485 KB	142 KB	70.72 %

40. LZ4: Wind Direction Data File (Process Data)

Source	File Size (Original)	File Size (Compressed)	Compression Ratio
Eixample	434 KB	8 KB	98.1 %
Sant Pol de Mar	372 KB	59 KB	84.1 %
Hospitalet	44 KB	2 KB	95.4 %
Raval	486 KB	144 KB	70.4 %
Sarria	486 KB	86 KB	82.3 %
Z. Universitaria	485 KB	144 KB	66.8 %

B. Source Code

In this experiments we obtained the source code from several sources. All the information can be accessed from the website below.

1. Huffman Code:
<http://code.activestate.com/recipes/577480-huffman-data-compression/>
2. Arithmetic Coding:
<https://www.drdoobs.com/cpp/data-compression-with-arithmetic-encodin>
3. LZ77:
<https://github.com/encode84/ulz>
4. LZ4:
<https://github.com/encode84/lz4x>