University of Alaska, Anchorage

College of Arts and Sciences

School of Engineering

Polar Compression Scheme

by

David Rathun

Supervisor:

Prof. Saif Zahir, PhD

A CAPSTONE PROJECT SUBMITTED TO THE SCHOOL OF ENGINNERING
AT UNIVERSITY OF ALASKA, ANCHORAGE FOR THE
DEGREE OF COMPUTER SCIENCE

Anchorage AK, May 2015

(c) Copyright 2015
by
David Rathbun

drathbu6@alaska.edu

Version 1.0

Abstract

The target of this capstone is to present a new compression scheme and suite that is not derived from previous notable schemes already established. To be brief, the suite strays away from using algorithms derived from Huffman and Lempel-Ziv. This restriction, or guideline in few cases, is to drive research into a completely different direction and develop new methodologies rather than improve upon current popular schemes and standards. Run-length, counting, and analytic encoding schemes are foundations of information theory, and will be the basis for all schemes within this project suite. Preliminary results have shown to improve the compression ratio of images while suffering greatly to attempts on data of text-based contents. Other possible and prospected outcomes have been discovered and noted, although not fully empirically tested as of this report.

Acknowledgments

"That moment when your program finally runs the way you want it to..." -Dustin Mendoza
"...and you have no idea why." -David Rathbun

This capstone is possible by the majority of help from Professor Saif Zahir as well as minor influences from colleagues such as Dustin Mendoza and others. Without them, perspectives would be few for the ideas that have come about and shaped into form. They are all an integral part in the group we share of our own academic courses, research, and projects.

Contents

1 Introduction 1

1.1 Introduction 1

1.2 Application 3

1.3 Motivation 6

1.4 Recent Developments 7

2 System Integration, Modeling, and Methodology 8

2.1 Java Implementation 8

2.2 Corpus & Comparison of Schemes 8

2.2.1 DEFLATE Scheme 9

2.2.2 LZMA Scheme 10

2.2.2 BZIP2 Scheme 10

2.2.3 RLE Implemented Scheme 10

2.3 Timeline 11

3 Design and Testing 12

3.1 Paper to Practice 12

3.2 Abstracting Utility Classes 12

3.3 Abstracting Schema Classes 13

3.4 New Developmental Algorithms 14

3.4.1 Mark and Exclude Algorithm 14

3.4.2 Rotary Tree 15

4 Results and Discussion 16

4.1 Measuring Prologue 16

4.2 (Developed Preprocessing Algorithms) 16

4.2.1 Preprocessing Algorithms 16

4.2.2 Simple Rotaries 17

4.2.3 Triangle Rotaries 19

4.2.4 Mark Exclusion Rotaries 22

4.3 Developed Compression Schemes 23

5 Summary and Conclusion 25

List of Figures & Tables

1.1	*Lossless compression scheme family tree.*	1
1.2.1	*Algorithm core applied to 4 bits*	3
1.2.2	*Rotary preprocessor applied to 4 bits*	5
1.3.1	*Cancellation differences*	6
2.2	*Calgary corpus visualized*	8
2.2b	*Random sample sizes generated from pseudo random numbers*	8
2.2.1	*DEFLATE Algorithm pathway*	9
2.3.1	*Rescheduled pathway timeline.*	11
3.3A	*FRLE Single-Pass Algorithm in Pseudo Code*	13
3.4A	*Mark and Exclude Compression Scheme*	14
3.4B	*Equivalence Rotary Scheme*	14
3.4C	*Equivalence Rotary Scheme on Non-Uniform Sparse*	14
3.4D	*Equivalence Rotary Scheme on Uniform Sparse*	15
3.4.1	*Rotary Tree Described*	15
4.2.3A	*Triangle Edges*	19

Chapter 1
Introduction

1.1 Introduction

Information theory[1] is comprised of many fields of study and applications, and primarily dictates the methods of how information is stored and transmitted. Keeping to a single area, compression and encoding, a new method will be devised that may be an addition to the already growing algorithms. This new method is rooted in the very foundation of the theory, and unlike popular schemes, it deviates from 'family pools' as soon shown. More specifically, it will not be familiar with Lempel-Ziv[2], described later, as it has been too heavily concentrated. It will also not be using Markovian[3] methods, also described later, as it grows too complex. Because of this, the goal of this new scheme is to hold no global header or vectors for the information it represents. Like many uncommon methodologies, this new scheme will be part of its own suite, with some unorthodox techniques.

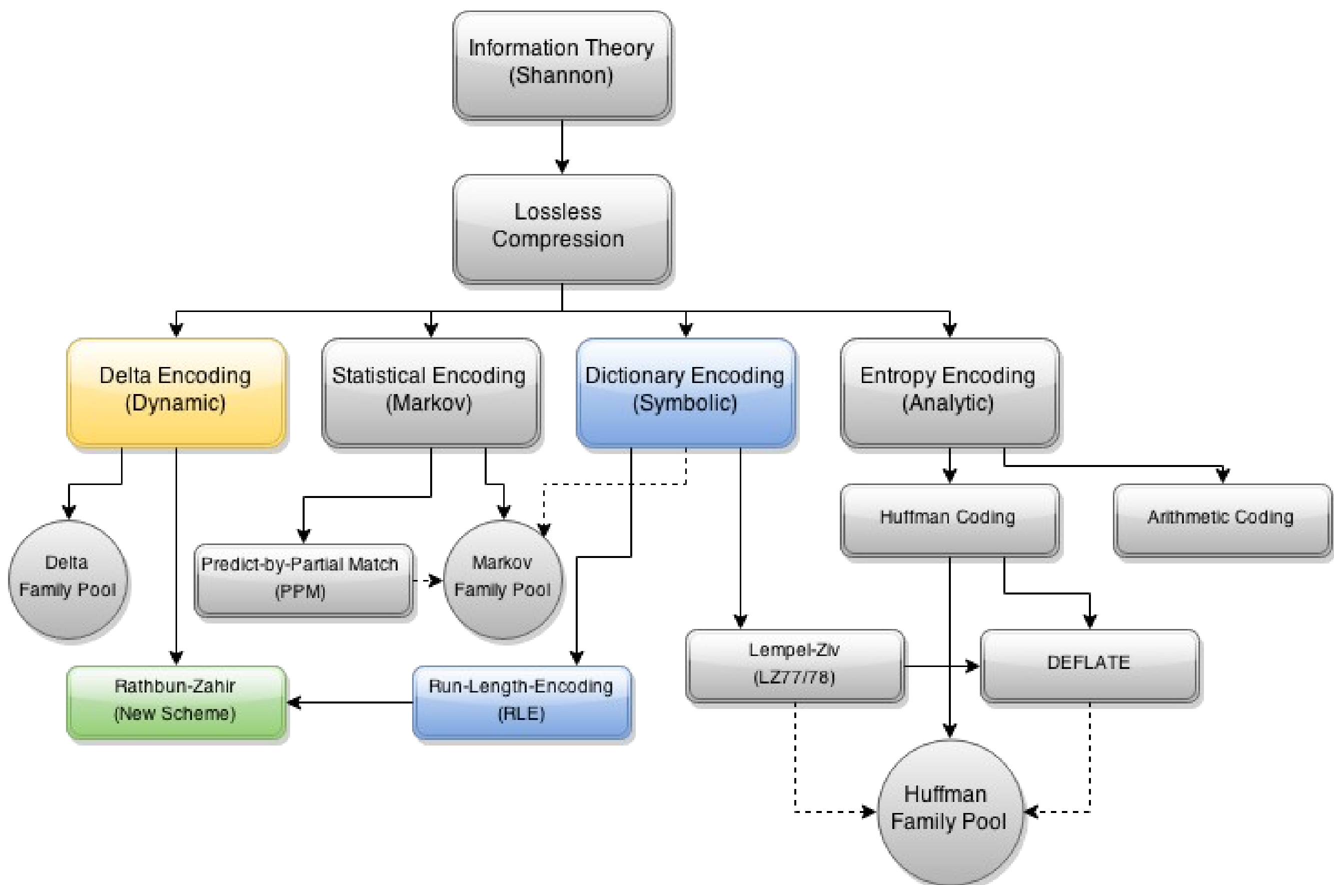


Fig 1.1 Lossless compression scheme family tree.

1 Claude E. Shannon, Warren Weaver. *The Mathematical Theory of Communication.* Univ of Illinois Press, 1949. ISBN 0-252-72548-4

2 US Patent No. 5532693 Adaptive data compression system with systolic string matching logic

3 A.A. Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons, 1971.

The family pools shown in Fig 1.1 are all indirectly connected to one another through various means, and although Lempel-Ziv (LZ) could have its own family pool, it is most implemented with the Huffman scheme, and Huffman has superiority over LZ for hierarchy of schemes. In fact, LZ at its core is nothing more than indexed run-length-encoding which will be described later.

Data encoding in general is done by reading a sparse, or stream, of information that has already been digitized to one's and zero's then discovering redundant patterns in which can be easily removed or truncated. This gives rise to compression, or deflating, the data itself as a whole. This also increases the entropy, or spreading information, thereby dictating that recursively applying a compression algorithm will result in dimensioning returns. Just like the second law of thermodynamics where more energy is required to decrease entropy, in the realm of lossless compression, all information has a finite absolute minimal length of representation, because compressing further requires more information to represent it (thereby decreasing entropy).

Compression schemes from the late 1940's onward have been playing on one another for decades. First there was Shannon-Fano that Huffman (one of Shannon's students) improved upon. Huffman coding was the defacto method for years until computing power increased as well as storage. The time came when Lempel-Ziv[4] (LZ77/78) introduced repetitive detection while encoding, greatly improving previous static compression schemes. The LZ scheme has now become the premier algorithm for general data compression, because it is empirically the fastest and highest compressor to use over a broad spectrum of data structures, streams, and storage. And because of this, there a hundreds of derivatives that start with the base form of the LZ method including the mergers of the Huffman methodology in most cases.

Many have tried to stray away from LZ, one of which is Markov-chain[5] methodology of encoding. Markov was not an information theorist, but a mathematician, and Markov was not the one to impose an encoding scheme in the name, it was actually Shannon. However, it was not implemented until the late 90's and it was actually combined first with LZ. However, it was so sound that other schemes implemented it without using the LZ scheme; especially for image processing and machine learning[6].

Our method, is one that will stay out of the pools. The flaw of LZ is of a fixed index size, and all data before (or after it) it must be known if an index points within it. A scenario such as bit-torrent technology or scattered streams would falter the method. And the primary flaw in Huffman is that all data must be known, be adapted, or the header must be known. The same is true with Markov-chains, especially as all states must be accounted for before hand. So the one way to fix this is to toss out these concepts as if they do not exist. If the input is a stream of data, so must be the output result, and can be easily tapped into at any 'block' position without the need for global header information.

4 Ziv, Jacob; Lempel, Abraham (May 1977). "A Universal Algorithm for Sequential Data Compression". *IEEE Transactions on Information Theory* **23** (3): 337–343.
5 Norris, James R. (1998). *Markov chains*. Cambridge University Press.
6 Baum, L. E.; Petrie, T. (1966). "Statistical Inference for Probabilistic Functions of Finite State Markov Chains". *The Annals of Mathematical Statistics* **37** (6): 1554–1563.

1.2 Application

The development and implementation of the new scheme will be done through joint cooperation between D. Rathbun and S. Zahir to further the data encoding technology. The primary application of this scheme are data streams, although it can be augmented for static data and images.

Core Algorithm:
The core of the scheme was created by restricting it to the following rules:

1. Scheme knows information already encoded.
2. Scheme knows information yet to be encoded.
3. Scheme knows information of the next single bit that was read.
4. Scheme does not know information beyond the next single bit that was read.
5. Scheme only compares the previous sparse directly with the next sparse.
6. Truncation, also known as cancellation, occurs on encoding the qualities of the comparison.

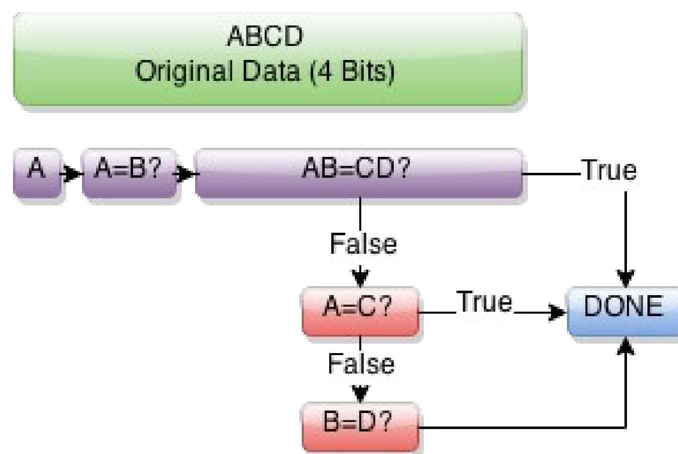


Fig 1.2.1 Algorithm core applied to 4 bits

The core algorithm as follows:

```
Boolean compare(data[], len, posA, posB)
Boolean o = true
for (i = 0 to len-1)
        o &= !(data[posA + i] XOR data[posB + I])
        if (!o) return false
return true;

Boolean compressDown(data[], len, posA, posB, out)
if compare(data, len, posA, posB)
        out.write(1)
        return true
else
        out.write(0)
        if (compressDown(data, len/2, posA, posB, out))
                compressCancel(data,len/2,posA+len/2, posB+len/2,out)
        else
                compressDown(data,len/2,posA+len/2, posB+len/2,out)
return false

Boolean compressCancel(data[], len, posA, posB, out)
if (compressDown(data, len / 2, posA, posB, out))
        compressCancel(data,len/2, posA+len/2, posB+len/2,out)
else
        compressDown(data, len / 2, posA + len / 2, posB + len / 2, out)
return false
```

To compress a full sparse or stream, a further complex algorithm is needed to be implemented, but the above code is the very heart of how the scheme operates.

Dynamic Algorithm:

The derivative of the core scheme adds the following rules:

1. The scheme knows itself, and uses its own process as information.
2. Multiple comparative faults may cause instant truncations.
3. Encoding must maintain or increase entropy, failure may result in catastrophic failure (no compression of the whole) or data thrown back to available preprocessing for another attempt.

The implementation of the dynamic algorithm is slightly more complex and will require further research into the scheme itself. The gist of the algorithm allows for a fault to occur when increasing length, but must not allow another comparative fault to occur when trying to propagate the cancellation immediately after the first fault. If it does, the algorithm restarts at the current position of the data and treats further as new data until such a scenario occurs again.

This may increase the compression ratio greatly but will create a flaw for data that repeats on a large scale. Therefor, the need for a preprocessing algorithm is not only recommended, but most likely required to reach a high compression ratio on general data. However, the preprocessing must follow the same restriction as the core algorithm. This means it can only read the next bit and use the previous data for encoding.

Preprocessing Algorithm:
A unique preprocessor scheme called the 'rotary' will also be researched and implemented to be contained in the suite. Its method is the same of the base algorithm, but has a 'moving window' of a single bit length that never expands.

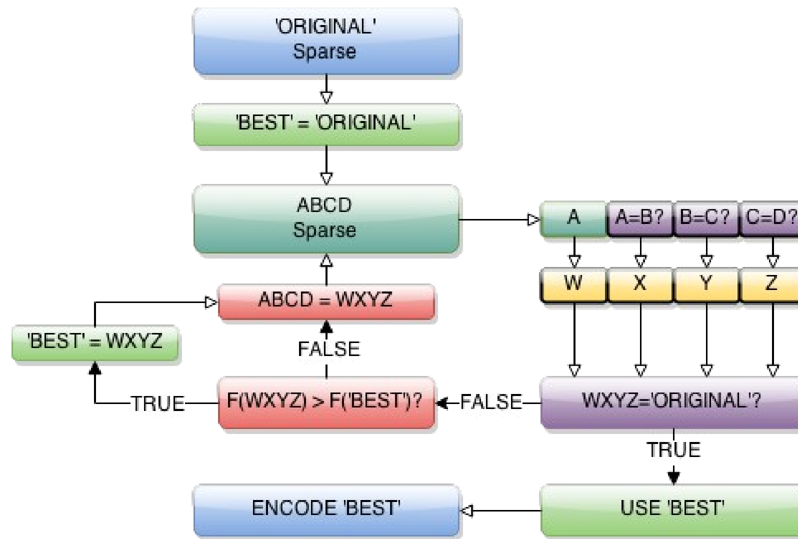


Fig 1.2.2 Rotary preprocessor applied to 4 bits

As demonstrated in Fig 1.2.2, the size of the information will always remain the same and applying the algorithm multiple times using the last result will eventually represent the exact original input. In this unique fashion the maximum quantitative possible rotaries can be defined as:

$$K_{overhead} = \left\lceil \log_2 (N_{length}) \right\rceil$$
$$R_{possibles} = 2^K$$

Rather than having a 'master' global header, the rotary index can be applied to either the sparse as a whole, each individual parts, or both. A further restriction to any global overhead required is defined as:

$$G_{overhead} \leq \sum 1 + \log_2 K_i$$

This restricts the global overhead to the most minimal length possible to indicate if rotaries are used and how they are used. Knowing the length of the rotary, will immediately indicate the maximum length of the sparse it represents. Another restriction can be applied and is carried over from the core algorithm:

$$N_{\text{subset}} = 2^x$$
$$N_{length} = \sum N_{\text{subset}}$$

This ensures that all subsets, or 'blocks' are 2^x in length and all rotary lengths definitively represent data length they have rotarized. There are exceptions, one being if the entire data has been rotarized since the information about its length is not needed as it ends when there is no further information to be read.

1.3 Motivation

The concept of the scheme was a simple ponder years ago by Rathbun. Beginning by simply modifying the run-length-encoding scheme at the bit level and having an unrestricted length of the symbol. In addition, it was to stray from overheads like indexing or table headers that is shared by so many popular schemes. Further pen-on-paper improvements were made such as truncation by cancellation; removing redundant information, if the information that is known already, by process of elimination.

Input 4 Bits	Output Symbol (Without Cancellation)	Output Symbol (With Cancellation)
0000	011	011
0001	01010	0101
0010	01001	01001
0011	01000	01000
0100	00010	0001
0101	001	001
0110	00000	00000
0111	00001	00001
1000	10001	10001
1001	10000	10000
1010	101	101
1011	10010	1001
1100	11000	11000
1101	11001	11001
1110	11010	1101
1111	111	111

Fig 1.3.1 Cancellation differences

However, the skill required to recursively implement it in code was staggeringly complex, at least on paper, as the cancellation required the algorithm to know itself.

The goal is to create a highly dynamic data structure that is not restricted to any one purpose and that can compete with other schemes already in place. Rather than have a header or indexing, the scheme will contain only the minimal overhead to represent the data itself and uses its own decoded length as an indication of completion.

1.4 Recent Developments

Given that this is a new scheme partially built on run-length-encoding (RLE), there has been minimal developments. RLE is considered an after-thought, or tool, or used as an example algorithm in the same fashion as 'bubble-sort' is used as an an example sorting algorithm. And because the core scheme is not shared by other schemes, this makes it an excellent candidate of being part of a new compression family.

The most recent development was the discovery of how to code the core algorithm recursively by splitting it into three parts: Compressing upwards by length multiplication that requires information from → compressing downwards by propagation then if it fails → by compress cancellation through truncation → by compressing downwards and so on. However, this is still a recursive process, it must be implemented into an iterative process which will require further research.

Chapter 2
System Integration, Modeling, and Methodology

2.1 Java Implementation

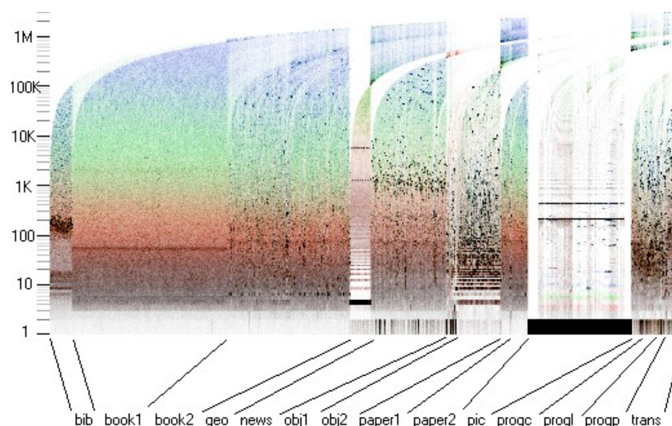
Although much of this encoding scheme is theoretical, it will need to be tested in the real world. The programming language for simple coding and testing will be done in Java, and will be used to interface with different file formats as sample data. Context of the sample data, or sparse arrays, may consist of binary, text, or readable binary strings of simple '1' and '0' characters. Java is excellent for this use as the usage of internal libraries will be prominent and the need for external libraries will be relatively low. Code will be transformed from pen and paper, into pseudo, and finally to Java.

2.2 Corpus & Comparison of Schemes

Another benefit to Java is the use internal compression scheme DEFLATE[1] which can be split into its classical format of LZ77+Huffman or Huffman only scheme through flag initialization. These two will be part of comparisons with the new scheme results. Others will include the following:

- LZMA[2] scheme using the SevenZip software development kit.
- BZIP2[3] using the Apache Commons Compress software development library.
- RLE[4] using simple implementations.

These were selected for popularity, diversity, and their general broad-based purposes that the new scheme also shares. They will all be bench-marked with two well-known corpus, the Calgary and the Canterbury as well as a pseudo random sample set based on the SHA2-512 hashing algorithm.



Source: http://mattmahoney.net/dc/dce.html

Fig 2.2 Calgary corpus visualized

---

1 Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996, http://www.rfc-editor.org/info/rfc1951

2 Pavlov I. (2015) LZMA SDK [Software Development Kit] Available from: http://www.7-zip.org/sdk.html

3 'Apache Commons' (2015) Apache Commons Compress [Software Development Library] Available from: http://commons.apache.org/proper/commons-compress/

4 Appendix C

By using two standard corpus specifically meant for compression, the uncertainty in pseudo random sparses for benchmarks will not be a primary factor, although it may affirm on real-world data. The pseudo random data will be generated by a rehash of each 64 bit value of the previous result, and input sizes trimmed to a prime number of bytes before being used as a sample to be compressed. The seed will be the 64-bit prime number '14249258496405449323' generated as the closest prime to: $\lfloor frac(\sqrt{\pi}) \rfloor * 2^{64}$. This will serve as the first 64-bit sample data. Example table as follows with the relative sizes:

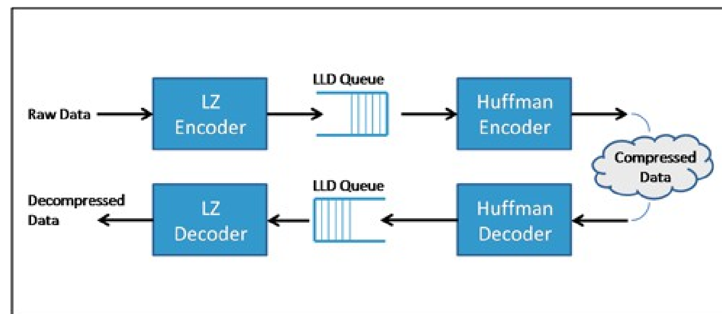
Bit Length	Byte Length	Prime Byte Length
64	8	7
512	64	61
4096	512	509
32768	4096	4093
262144	32768	32749
2097152	262144	262139
16777216	2097152	2097143

Fig 2.2b Random sample sizes generated from pseudo random numbers

In theory, no compressor *should* be able to decrease the original size. Assuming this, all results with a ratio nearest to 1, or in astronomically rare cases, lower will be deemed better than those with a greater number.

2.2.1 – DEFLATE Scheme

Although combined with Huffman coding, DEFLATE is most closely related to the original Lempel-Ziv compression scheme that still survives. Originally developed and patented by PKZIP, however, the current mass distributed version is actually an open-source interpretation from ZLIB software development library. The scheme is composed of two stages. The first uses the classic LZ77 dictionary look-back encoding and compressor. The second stage attempts to reduce the compressed length further by using Huffman encoding. The end result is highly compressed, can be quickly decoded, as well as universally known by many software applications without needed overhead.



Source: http://www.chipestimate.com/tech-talks/2010/03/23/Altior-Unzipping-the-GZIP-compression-protocol

Fig 2.2.1 DEFLATE Algorithm pathway

Java's internal libraries provide a method to disable the LZ77 stage and use only Huffman coding. This gives an easy benefit as to add an extra comparison to the overall results, and will be used as such.

2.2.2 – LZMA Scheme

Another common scheme is Lempel-Ziv-Markov-chain-Algorithm or LZMA for the much needed shortened name. It was original developed by Igor Pavlov and first seen implemented by 7Zip software also connected to Pavlov. The scheme uses the ultimate, by opinion, hybrid of LZ77 and Markovian methodology where the input is allowed to be massive, the dictionary between tiny and huge, and the results to be smaller than other popular methods. It is highly complex and dynamic where everything is encoded including the parts of itself.

The first stage is a modified LZ77. As with LZ77, the scheme uses locational index dictionary for repeating areas, however its range is done through range encoding, and storage of both indexes and ranges are done at bit level and cached by hash chaining. The current form of itself competes easily with other schemes, even so much as to make them inferior on most general cases[5].

2.2.2 – BZIP2 Scheme

Unlike other schemes, BZIP is a Run-Length derivative using a multistage process that includes the creation of Burrows–Wheeler[6] transforms and multitudes of Huffman tables. Although the scheme is simple to understand, the individual stages add complexity. The early stages are purely designed to help make RLE be more effective, while the last stages are clean-up and final compression via Huffman.

Because of it not using LZ77, it has the added benefit of being the token 'not LZ' candidate for the result comparisons. As the new scheme is also somewhat derived from RLE, this makes it an excellent benchmark.

2.2.3 – RLE Implemented Scheme

Unfortunately, there is not set standard for RLE as it is a methodology, and the symbol fields are coded in the eyes of the beholder. Three implementations will be made to be used as comparisons with the overall results.

Optimal – Bit level run-length with runs stored as three bits. Best cases reduce the size by half, worst cases will increase data over three times the original size. Chosen as to be most similar to the new scheme's best result ratio.

Half-Size **–** Nibble (four bit) level run-length with runs stored as four bits. Closely related to traditional 2-Byte RLE. Best cases reduce the data to below 1/32 of the original size. Worst cases will never exceed over two times the original size.

Sparse Packed **–** Byte level run-length with only sparse runs stored as a 1 byte length symbol. Popular optimized method of 2-Byte RLE. Sparses '0xFF' and '0x00' are the only bytes that are encoded with their run-length.

---

5 History of Lossless Data Compression Algorithms. (n.d.). Available from http://www.ieeeghn.org/wiki/index.php/History_of_Lossless_Data_Compression_Algorithms

6 Burrows, Michael; Wheeler, David J. (1994), A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation

2.3 Timeline

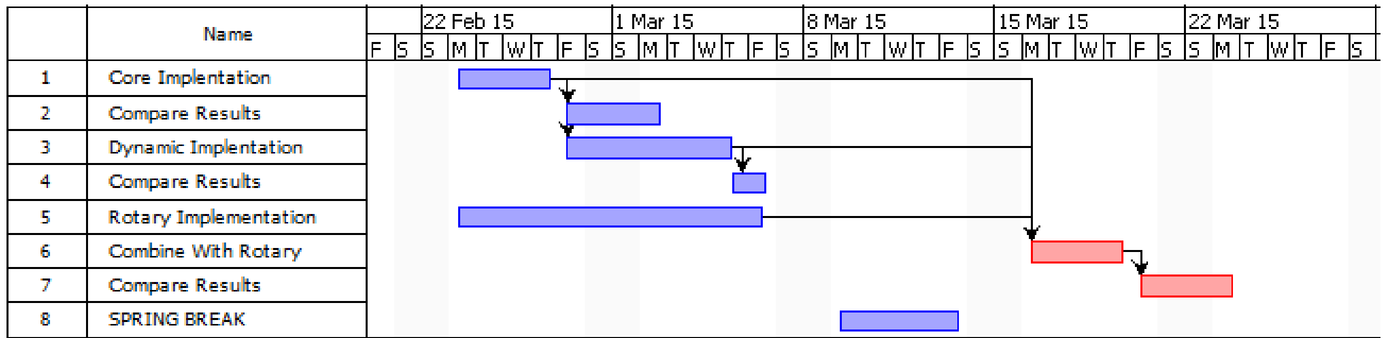


Fig 2.3.1b Rescheduled pathway timeline.

Chapter 3
Design and Testing

3.1 Paper to Practice

Through the Java programming language, modular coding has been devised to create a simplistic testing environment for the different schema. Four new classes called [BitInputStream], [BitOutputStream], [Explode], and [Implode] have been coded to handle bit-level encoding and decoding when working with the different algorithms as well as different sparse types. All schemes have been abstracted out to a respective class when testing.

3.2 Abstracting Utility Classes

Backward compatibility and redundancy was needed when traversing individual bits. The [Explode] class for it name sake is used to convert a byte array into a represented bit array as another byte array itself. However, this obviously increases the size of the input and working data to eight times the original size. The [Implode] class does the exact opposite, by converting the sometimes oddly ended bit array to a byte array equivalent that may need padding on the end.

When working with the Java-Development-Kit (JDK) library based [Stream] classes, two non-extended wrapper class will be used to directly handle large data at near byte-level. The [BitInputStream] reads bits one by one where algorithms can flow easily through without using “exploded” bit arrays. The [BitOutputStream] class writes directly to any [ByteArrayOutputStream] object. Both classes are used to rapidly test preprocessing algorithms.

Other utility classes come from the [DiscreteBytes] library developed by Rathbun (the author) for common crypto-hash, random, and string representation utilities. Notable algorithms used come from the Secure-Hash-Algorithm (SHA2-256) and the implemented Marsaglia[1] “xorshift” random number generator, a suitable replacement for JDK randomization until it was improved with the same algorithm in Java version 8.

1 Marsaglia, George (July 2003). "Xorshift RNGs". Journal of Statistical Software 8 (14).

3.3 Abstracting Schema Classes

The current developmental algorithm, PRZ, has been abstracted to use the “exploded” byte arrays for debugging purposes. The two schemes called Core and Dynamic have independent classes. The Rotary class is also independent, yet uses the [BitInputStream] and [BitOutputStream] for speed purposes. For the PRZ scheme to be used, bytes must be “exploded” into a bit array before being encoded.

The Lempel-Ziv (LZ77, LZ78) algorithm is only as good when paired with Huffman. For this, the DEFLATE[2] library from JDK is used as it is also an extended [Stream] class. DEFLATE also can produce raw Huffman encoding as a standalone compressor and will be used as such.

The Lempel-Ziv-Markov-chain-Algorithm (LZMA)[3] scheme will also be abstracted out into a workable class. Given that it is from SevenZip library, it is the most cumbersome to abstract into simple utility coding.

The BZIP2[4] from the Apache Commons Compress software development library uses the [Stream] class and therefor was abstracted without issue.

Run-Length-Encoding (RLE) has been implemented using a recursive bilateral algorithm. Although three different implementations were planned, only one is used called Focused, or Fast, Run Length Encoding (FRLE). Because PRZ is also run-length based, FRLE uses the same preprocessor Rotary as a benchmark for comparison. Since there are many interpretations of RLE, the implemented two-bit symbolic FRLE algorithm is described below.

```
lastBitRead = read()
// Possible values are '0', '1', and terminator '-1'
while (lastBitRead >= 0):
      middleBit = read()
      if ((middleBit = lastBitRead) AND (middleBit NOT -1)):
            peekBit = peek()  //peek at next bit
            if ((middleBit = peekBit) AND (peekBit NOT -1)):
                  write(middleBit)     // Write two-bit symbol
                  middleBit = read()   //  for the three
                  write(middleBit)     //   matching bits.
                  lastBitRead = read() //prep next bit
                  continue    // continue loop
            end if
      end if
      write(lastBitRead)       //attempt failed, write
      write(lastBitRead XOR 1) //two-bit symbol for one bit
      lastBitRead = middleBit
loop
```

Fig 3.3A: FRLE Single-Pass Algorithm in Pseudo Code

---

2 Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996, http://www.rfc-editor.org/info/rfc1951

3 Pavlov I. (2015) LZMA SDK [Software Development Kit] Available from: http://www.7-zip.org/sdk.html

4 'Apache Commons' (2015) Apache Commons Compress [Software Development Library] Available from: http://commons.apache.org/proper/commons-compress/

3.4 New Developmental Algorithms

Sometimes the most influential discoveries are made by mistake, such as the case of another algorithm created from improving the Rotary. Implementation and testing has shown to not only better compress than PRZ but also in few cases much of the other schemes. However, because it is not the focus of this project, it will only be noted and used as a benchmark. Due to its behavior as being derived from the Rotary, it has be duly named “Mark and Exclude” algorithm.

3.4.1 – Mark and Exclude Algorithm

```
Data    : 1001 1000 0001 1111 0101 1001 1001 1000
Mark    :         1         0         0         1
Xor 8Bit: XXXX XXXX 1000 0111 1100 0001 XXXX XXXX
Mark    :              1    0    0    0
Xor 4Bit: XXXX XXXX XXXX 1111 XXXX 1001 XXXX XXXX
Mark    :                   1 1       0 0
Xor 2Bit: XXXX XXXX XXXX XXXX XXXX 0110 XXXX XXXX
Mark    :                          0100

Final: 1001 1000 1000 1110 0110 0011 0001 00
```

Fig 3.4A: Mark and Exclude Compression Scheme

The above scheme discovery was based on the following alternative Rotary improvement that attempts to transform the data in $\Theta(n\log_2(n))$ time.

```
Data    : 0001 1000 0000 1111 0101 1001
Xor 8Bit: 0001 1000 0001 0111 0100 0001
Xor 4Bit: 0001 1001 0000 0110 0101 0000
... Done ...
Final   : 0001 1001 0000 0110 0101 0000
```

Fig 3.4B: Equivalence Rotary Scheme

Notice the highly repetitive groupings. This can be done on data that is not as uniform...

```
Data    : 1001 0110 0101 1001 0000 1000
Xor 8Bit: 1001 0110 1100 1111 1001 0001
Xor 4Bit: 1001 1111 0101 0110 0000 1000
Xor 2Bit: 1011 0101 1111 1100 1010 0010
... Done ...
Final   : 1100 1010 0000 0011 0101 1101
```

Fig 3.4C: Equivalence Rotary Scheme on Non-Uniform Sparse

This can also be done on data that is highly uniform...

```
Data    : 1110 1110 1110 1110 0000 0000
Xor 8Bit: 1110 1110 0000 0000 1110 1110
Xor 4Bit: 1110 0000 1110 1110 0000 0000
Xor 2Bit: 1101 1111 0001 0001 1111 1111
Xor 1Bit: 1010 0000 1110 1110 0000 0000
Final   : 1010 0000 1110 1110 0000 0000
```

Fig 3.4D: Equivalence Rotary Scheme on Uniform Sparse

As shown in Fig 3.4D, even though the data was repeatable before, it has kept much of the repetitions. However, this alternative Rotary has been superseded by a fixed-sized index rotary which uses trees as discussed in next section.

3.4.2 – Rotary Tree

Through Boolean algebra, patterns emerged in the Rotary algorithm itself. Three unique selectable bit-streams derived from the original data have shown to be of certain cases in which improve compression for all schemes.

```
Data         : 1000 1011 1011 1010

                     Bottom Edge
Normal Rotary: 1011 0001 1001 1000
Tree Build..    001 0110 1010 1011
Tree Build.. L   10 0010 0000 0001
Tree Build..  e   0 1100 1111 1110 Right Edge
Tree Build..   f    0101 0111 1110
Tree Build..    t    000 0011 1110
Tree Build..          11 1101 1110
Tree Build..      E    1 1100 1110
Tree Build..       d     1101 0110
Tree Build..        g     100 0010
Tree Build..         e     01 1100
Tree Build..                0 1101
Tree Build..                  0100
Tree Build..                   001
Tree Build..                    10
Tree Build..                     0

Bottom Edge: 1011 0001 1001 1000
Left Edge  : 1010 0011 1100 0010 ← Best Selection
Right Edge : 0110 0000 0001 0100 ← Worst Selection
Reverse LE : 0100 0011 1100 0101
Reverse RE : 0010 1000 0000 0110
```

Fig 3.4.1A: Rotary Tree Described

Chapter 4
Results and Discussion

4.1 Measuring Prologue

Two descriptors are used to determine the overall characteristics of the data without requiring a human description other than the file's original given name. First, the measurement of the single-order entropy is used for the sake of being a familiar formula in information theory. The measurement of entropy is defined as:

Entropy

$$H(X) = -\left(\mathrm{P}(0_{bit}) \log_2 \mathrm{P}(0_{bit}) + \mathrm{P}(1_{bit}) \log_2 \mathrm{P}(1_{bit}) \right)$$

The second measurement is arbitrarily named "density" and is used as an alternate descriptor of the data. For this measurement, the "Derivative" Rotary described in 4.A is iterated at least 4 times and the smallest value out of all results is used. It is generally defined as:

Density

$$P_D(x_{bit}) = \frac{count_x(derive(bits_{data}))}{N_{\mathrm{length}}}$$

$$H_D(X) = -\left(P_D(0_{bit}) \log_2 P_D(0_{bit}) + P_D(1_{bit}) \log_2 P_D(1_{bit}) \right)$$

Density is somewhat proportional to the compressibility and a general predictor to the outcome. It is not proportional to the entropy, only to the discrete pattern repetitions of neighboring bits. As such, when data is described as "dense" is generally both at high, or near full, entropy and randomness. Consequently, although textual data is humanly perceived as low entropy, it is in fact quite the opposite when measured at the bit level order. Pattern recognition algorithms are not affected due to primarily per-processing and compressing at the byte level of input, where entropy is much lower.

As a side note, density at the byte level can only be determined by splitting each byte into eight parallel streams, where each stream represents the bit position of each byte and the values at such position. The average and standard deviation of each measured stream density would be the descriptor. However, this method is not used, due to multiple pass algorithms and variable bit field algorithms where the alignment is not necessarily at the byte level.

4.2.1 Preprocessing Algorithms

Although some preprocessing algorithms were attempted, only the "Rotary" variants were used for the new compression algorithm variants. Move-To-Front (MTF)[1] and Burrows-Wheeler (BWT)[2] transforms were considered and tested, but were discarded due to poor or negligible differences in the results. Because the Rotary is of the same symmetry of the new compression schemes, it was decided to be exclusively used.

1 Ryabko, B. Ya Data compression by means of a "book stack", Problems of Information Transmission, 1980, v. 16: (4), pp. 265–269

2 Burrows, Michael; Wheeler, David J. (1994), A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation

4.2.2 Simple Rotaries

Simple Rotary: Classic Derive – 'Rotarize' or 'Triangle Rotary: Bottom Edge'

This attempts to decrease overall entropy and density, as a smoothing method, through multiple rounds. Maximum possible rounds is length N.

```
define LAST_BIT as BIT
define OUT as BITFIELD of length DATA_BITLENGTH
LAST_BIT = DATA_BITS[0]
OUT[0] = LAST_BIT
for i = 1 to (DATA_BITLENGTH - 1)
    OUT[i] = LAST_BIT XNOR DATA_BITS[i]
    LAST_BIT = DATA_BITS[i]
next
return OUT
```

Example:

```
Input:   01101000 11110000 11110000 00001111 ← H(x) = 0.997
Round 1: 00100011 01110111 01110111 11110111 ← H(x) = 0.896
Round 2: 01001101 00110011 00110011 11110011 ← H(x) = 0.989
Round 3: 00010100 01010101 01010101 11110101 ← H(x) = 1.000
Round 4: 01100000 00000000 00000000 11110000 ← H(x) = 0.696
Round 5: 00101111 11111111 11111111 01110111 ← H(x) = 0.625
```

Simple Rotary: Classic Derive Improved – 'Derive'

Like the Classic Derive, this decreases overall Entropy and Density where time is not a factor. Maximum possible rounds is length $2N$.

```
define LAST_BIT as BIT
define OUT as BITFIELD of length DATA_BITLENGTH
LAST_BIT = DATA_BITS[0]
OUT[0] = LAST_BIT XOR 1
for i = 1 to (DATA_BITLENGTH - 1)
    OUT[i] = LAST_BIT XNOR DATA_BITS[i]
    LAST_BIT = DATA_BITS[i]
next
return OUT
```

Example:

```
Input:   01101000 11110000 11110000 00001111 ← H(x) = 0.997
Round 1: 10100011 01110111 01110111 11110111 ← H(x) = 0.857
Round 2: 00001101 00110011 00110011 11110011 ← H(x) = 0.997
Round 3: 11110100 01010101 01010101 11110101 ← H(x) = 0.974
Round 4: 01110000 00000000 00000000 11110000 ← H(x) = 0.758
Round 5: 10110111 11111111 11111111 01110111 ← H(x) = 0.544
```

Simple Rotary: 'Classic Integrate'

This attempts to increase Density while attempting to decrease higher order Entropy. Also the direct inverse of 'Classic Derive'. Maximum possible rounds is length N .

```
define LAST_BIT as BIT
define OUT as BITFIELD of length DATA_BITLENGTH
LAST_BIT = DATA_BITS[0]
OUT[0] = LAST_BIT
for i = 1 to (DATA_BITLENGTH - 1)
    OUT[i] = LAST_BIT XNOR DATA_BITS[i]
    LAST_BIT = OUT[i-1]
next
return OUT
```

Example:

```
Input:   01101000 11110000 11110000 00001111 ← H(x) = 0.997
Round 1: 00011010 00001010 00001010 10100000 ← H(x) = 0.857
Round 2: 01000110 10100110 10100110 01101010 ← H(x) = 0.997
Round 3: 00101110 01101110 01101110 11100101 ← H(x) = 0.954
Round 4: 01100001 00011110 11100001 11101100 ← H(x) = 0.997
Round 5: 00010100 10111110 00010100 00011101 ← H(x) = 1.000
```

Simple Rotary: Improved Classic Integrate – 'Integrate'

Like the Classic Integrate, although a more “greedy” method. May not produce best results but will produce fast good results. Also direct inverse of 'Derive'. Maximum possible rounds is length $2N$.

```
define LAST_BIT as BIT
define OUT as BITFIELD of length DATA_BITLENGTH
LAST_BIT = DATA_BITS[0] XOR 1
OUT[0] = LAST_BIT
for i = 1 to (DATA_BITLENGTH - 1)
    OUT[i] = LAST_BIT XNOR DATA_BITS[i]
    LAST_BIT = OUT[i-1]
next
return OUT
```

Example:

```
Input:   01101000 11110000 11110000 00001111 ← H(x) = 0.997
Round 1: 11100101 11110101 11110101 01011111 ← H(x) = 0.857
Round 2: 00010011 11110011 11110011 00111111 ← H(x) = 0.928
Round 3: 10110111 11110111 11110111 01111111 ← H(x) = 0.625
Round 4: 01110000 00001111 11110000 11111111 ← H(x) = 0.974
Round 5: 11110101 01011111 11110101 11111111 ← H(x) = 0.696
```

4.2.3 Triangle Rotaries

Through more development, there was a need to decrease Entropy and Density rapidly on first try through an $O(N)$ or $O(N+\frac{N}{\log_2 N})$ algorithm. The basic three are the Left Edge, and Right Edge, and Bottom Edge also known as "Classic Derive." While the bottom edge can easily run in $O(N)$ it was not a suitable predictor for the existence of a "good" rotary index.

The Left Edge describes the overall propagation, however, it cannot predict the quality of the rotary indices, as it will miss potential low entropy areas residing internally. This edge can indicate the chances of finding a "good" rotary.

The Right Edge describes the entire data as a whole. The only flaw is the decoding taking at least $O(2N+\frac{N}{\log_2 N})$ time. However, this risk outweighs the benefits as the output is an instantaneous "good" rotary due to patterns propagating continuously to the left. Another few rounds of Classic Derive immediately after should find a better result, at least behaviorally.

Furthering importance and the value of the Left and Right Edges, the Left Edge was discovered, quite late, to be able to run in $O(N)$ and could not be currently taken advantage on. The Right Edge, given the greatest reward still eludes a proper algorithm.

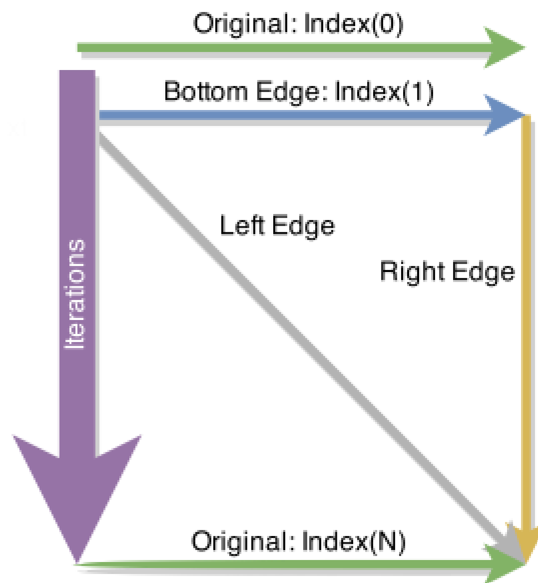


Fig. 4.2.3A Triangle Edges

Triangle Rotary: 'Left Edge' or 'Silver Edge'

There is no pseudo code as the required pen-and-paper implementation is to run 'Classic Derive' $O(\frac{N^2}{2})$, and retrieved the Nth bit of each round. Properties of the resulting bit field is unique from the input data, as it is not part of a simple predictable rotary. A hypothetical property is if the outcome does not decrease the density, the Right Edge or Bottom Edge is guaranteed to do so, however, Right Edge may require as many rounds to complete.

Below is an example of how bits propagate, as highlighted yellow, in the simple rotary of Classic Derive and the representative Left Edge, highlighted in blue. The left area of bits is omitted as they are not required to determine the resulting sequence.

Example:

```
    Input: 01101000 11110000 11110000 00001111 ← H(x) = 0.997
    Prep 1:   00100011 01110111 01110111 11110111 ← Triangle Bottom
    Prep 2:   .1001101 00110011 00110011 11110011
    Prep 3:   ..010100 01010101 01010101 11110101
    Prep 4:   ...00001 00000000 00000000 11110000
    Prep 5:   ....1110 01111111 11111111 01110111 ← Triangle Bottom
    Prep 6:   .....110 00111111 11111111 00110011
    Prep 7:   ......10 11011111 11111111 01010101
    Prep 8:   .......0 01001111 11111111 00000000
    Prep 9:   ........ 10010111 11111111 01111111 ← Triangle Bottom
    Prep 10:  ........ .0100011 11111111 00111111
    Prep 11:  ........ ..001101 11111111 01011111
    Prep 12:  ........ ...10100 11111111 00001111
    Prep 13:  ........ ....0001 01111111 01110111
    Prep 14:  ........ .....110 00111111 00110011
    Prep 15:  ........ ......10 11011111 01010101
    Prep 16:  ........ .......0 01001111 00000000
    Prep 17:  ........ ........ 10010111 01111111 ← Triangle Bottom
    Prep 18:  ........ ........ .0100011 00111111
    Prep 19:  ........ ........ ..001101 01011111
    Prep 20:  ........ ........ ...10100 00001111
    Prep 21:  ........ ........ ....0001 11110111 ← Triangle Bottom
    Prep 22:  ........ ........ .....110 11110011
    Prep 23:  ........ ........ ......10 01110101
    Prep 24:  ........ ........ .......0 10110000
    Prep 25:  ........ ........ ........ 00010111
    Prep 26:  ........ ........ ........ .1100011
    Prep 27:  ........ ........ ........ ..101101
    Prep 28:  ........ ........ ........ ...00100
    Prep 29:  ........ ........ ........ ....1001
    Prep 30:  ........ ........ ........ .....010
    Prep 31:  ........ ........ ........ ......00
    Prep 32:  ........ ........ ........ .......1

Final:     01001110 10010110 10010110 01101001 ← H(x) = 1.000
```

Triangle Rotary: 'Right Edge' or 'Golden Edge'

As with the Left Edge, there is no pseudo as the required pen-and-paper implementation is to run 'Classic Derive' $O(\frac{N^2}{2})$ and retrieved the last bit of each round. Properties of the resulting bit field is unique from the input data, as it is not part of a simple predictable rotary. A hypothetical property is if the outcome does not decrease the density, the Left Edge or Bottom Edge is guaranteed to do so.

Example:

```
   Input:    01101000 11110000 11110000 00001111 ← H(x) = 0.997
   Prep 1:   00100011 01110111 01110111 11110111 ← Triangle Bottom
   Prep 2:   .1001101 00110011 00110011 11110011
   Prep 3:   ..010100 01010101 01010101 11110101
   Prep 4:   ...00001 00000000 00000000 11110000
   Prep 5:   ....1110 01111111 11111111 01110111 ← Triangle Bottom
   Prep 6:   .....110 00111111 11111111 00110011
   Prep 7:   ......10 11011111 11111111 01010101
   Prep 8:   .......0 01001111 11111111 00000000 (+1111111)
   Prep 16:  ........ ........ ....1111 00000000 (+1111111)
   Prep 24:  ........ ........ .......1 00000000 (+1110100)
   Prep 32:  ........ ........ ........ .......1

Final:    11101110 11111110 11111110 11101001 ← H(x) = 0.758
```

4.2.4 Mark Exclusion Rotaries

Mark Rotary: Single Conquer – 'Exclude'

This variant was used in conjunction with an older Mark & Exclude (M&EC) algorithm and shares similarity to the internal process of its hybrid (HYBC). It is only worth noting as it was a single-index rotary, thereby not needing overhead. Its purpose was to improve the divide and conquer behavior that the M&EC applies.

```
define mask8 = byte[0]
define mask4 = byte[0] >> 4
define mask2 = byte[0] >> 6
define mask1 = byte[0] >> 7

mask2 = mask2 XOR ((mask1 << 1) OR mask1))
mask4 = mask4 XOR ((mask2 << 2) OR mask2))
mask8 = mask8 XOR ((mask4 << 4) OR mask4))

byte[0] = byte[0] XOR (mask8 AND 0x7F)
for i = 0 to N
    byte[i] = byte[i] XOR mask8
next
```

Mark Rotary: Counting Divide & Conquer

This variant is used internally in the Mark and Exclude (M&EC) algorithm. The pseudo code would be quite long, and would be simpler to describe. The two largest counts at the byte level are taken and stored. When running at this level, should any comparisons fail, the byte is XOR'd with the highest count byte. This process continues until a full round fails all comparison checks and the next phase is the four-bit level. This level is not rotarized, and when finished continues to the last phase which is 2 bit. Just like the eight-bit level, the highest count is taken and stored, and all comparison fails are XOR'd with the highest 2 bit count.

Mark Rotary: Neighbor Divide & Conquer

This variant is used internally in the Hybrid Mark and Exclude (HYBC) algorithm. Like the Counting Divide & Conquer, it shares a similar process, however it does not find highest counts, only last known failed neighbor. This provides no overhead needed, although very similar results with other Mark Rotaries.

4.3 Developed Compression Schemes

Polar Schemes

These schemes do not rely on foundation schemes such as Lempel-Ziv or Huffman, nor do they rely on any global header required by those families and many others. Because of this, the behavior is a stream format sharing only similarities with common Run-Length-Encoding.

Polar Scheme: Core – 'CPRZ'

The original compressor. The window doubles at each 2^n position, and compares the previous 2^n bit field with the next 2^n bit field. If there is not enough data, the window resets to 1 and continues the rest as if it was new data until it occurs again. Comparison faults result in 'cancellation' propagation that attempts to salvage any repetitions possible. Best resulting length cases for any data is $\log_2 N+1$, or simply $n+1$ in base two as a block. On the other side of the spectrum, the worst cases result in $2N-\log_2 N-1$, or 2^n-n-1 in base two as a block. Pseudo code as follows.

```
int coreUp(bit[] bits, int length) {
     int compLen = 1;
     int posA = 0;
     int posB = 1;
     write(bits[posA]);
     while (posB < length) {
          coreDown(bits, compLen, posA, posB);
          compLen *= 2;
          posB = posA + compLen;
          if (posB + compLen > length) {
               posA = posB;
               if (posA >= length) return posA;
               posB = posA + 1;
               compLen = 1;
               write(bits[posA]);
}    }    }

int coreDown(bit[] data, int compLen, int posA, int posB) {
     if (compLen < 1) return -1;
     if (compare(data, compLen, posA, posB)) {
          write(1);
          return 1;
     } else {
          write(0);
          if (compLen > 1) {
               int split = compLen / 2;
               if (coreDown(data, split, posA, posB) > 0) {
                    coreCancel(data, split, posA + split, posB + split);
               } else {
                    coreDown(data, split, posA + split, posB + split);
     }    }    }
     return 0;
}

int coreCancel(bit[] data, int compLen, int posA, int posB) {
     if (compLen > 1) {
          int split = compLen / 2;
          if (coreDown(data, split, posA, posB) > 0) {
               coreCancel(data, split, posA + split, posB + split);
          } else {
               coreDown(data, split, posA + split, posB + split);
          }
     }
     return 0;
```

```
}
```

Polar Scheme: Dynamic – 'DPRZ'

In this scheme, the running window resets if the first comparison faults when propagating after the first failed attempt to grow the window. Threshold for this failure is set at the 4-Bit versus 4-Bit comparison and higher. As with the Core algorithm, the best and worst case lengths are the same.

Mark Schemes

These algorithms rely on all data being available in their entirety. All behave in a divide and conquer fashion with internal rotarization. The final output is often unitary formed and therefor can be easily compressed with the Dynamic algorithm on the finalization.

Mark Scheme: Mark & Exclude Chain – 'M&EC'

This algorithm shares a similarity with Huffman, although it only takes the two highest counts and then compares, excludes, divides, rotary, and conquers the entire bit-field, it then uses the Dynamic algorithm to compress the header of comparisons bit values. Unlike the Core variants, this algorithm requires the entire data sparse to be known. Best results are in the similar range of $\log_2 N + C$ while worst results are near $\frac{3N}{2} + C$. This algorithm is only worth mentioning as it performed better than the Core and Dynamic in text and object data. However, due to the close behavioral similarities with Huffman, it will be further omitted from development to prevent coincidental bias.

Mark Scheme: Hybrid Mark & Exclude Chain – 'HYBC'

Unlike the previous scheme, this algorithm does not take the highest counts, but uses the last known comparison bit sequence at the current length to compare the current sequence. Although now without the behavior of Huffman, it performs much poorly in most text and object data.

Run Length Encoding Scheme: Fast RLE – 'FRLE'

Rather than using a byte-level run length encoding, a bit-level with similar behavior to that of the Core algorithm was chosen. This algorithm is simply for the sake of being a benchmark and primary goal to overcome with Huffman likewise being the second. The algorithm is simple, if a single bit repeats two more times it is marked with two bits, if not, it is also marked with the converse representation of the two bits. Therefor, best case result is $\frac{2N}{3} + C$ while worst is $2N + C$. To prevent such a large worst case it is reduced to $N + 1 + C$ if failed to compress, as any modern scheme does, as shown in the results.

					Ratio = (New Size / Original Size)								
Corpus	Name	Bits	H(X)	$H_D(X)$	CPRZ	DPRZ	M&EC	HYBC	FRLE	HUFF	LZHF	BZIP	LZMA
Zahir	img182.bmp	403696	0.242	0.031	0.0730	0.0470	0.0404	0.0620	0.6689	0.1344	0.0046	0.0020	0.0022
	source1.bmp	25008	0.787	0.036	0.1044	0.0477	0.0585	0.0409	0.6692	0.1635	0.0240	0.0336	0.0234
	img29.bmp	112816	0.338	0.058	0.1429	0.0878	0.0623	0.1071	0.6758	0.1403	0.0071	0.0068	0.0062
	img5.bmp	990672	0.818	0.077	0.1803	0.1302	0.1780	0.1769	0.6780	0.1872	0.0641	0.0606	0.0511
	img180.bmp	74096	0.845	0.087	0.1823	0.1481	0.1906	0.2008	0.6814	0.2048	0.1106	0.0949	0.0902
	img210.bmp	322288	0.955	0.091	0.1969	0.1363	0.2453	0.1943	0.6822	0.2022	0.0813	0.0817	0.0609
	img168.bmp	224176	0.983	0.103	0.2305	0.1496	0.2683	0.2111	0.6878	0.2139	0.0868	0.0809	0.0729
	img4.bmp	303088	0.787	0.111	0.1826	0.1697	0.2274	0.2326	0.6857	0.2049	0.0797	0.0851	0.0629
	img105.bmp	121136	1.000	0.118	0.2023	0.1457	0.2665	0.1959	0.6909	0.2447	0.0975	0.1032	0.0872
	img199.bmp	123344	0.918	0.133	0.2745	0.1861	0.2835	0.2592	0.6909	0.2415	0.1394	0.1425	0.1216
	img53.bmp	73648	0.736	0.137	0.2948	0.2211	0.2293	0.3094	0.6966	0.2131	0.1248	0.1236	0.0981
	img169.bmp	194480	0.984	0.143	0.3147	0.2371	0.3145	0.3359	0.6902	0.2481	0.1259	0.1128	0.0915
	img26.bmp	263152	0.884	0.157	0.3100	0.2362	0.2814	0.3298	0.6980	0.2427	0.1398	0.1434	0.1019
	img75.bmp	79536	0.895	0.160	0.2946	0.2275	0.3014	0.3161	0.7009	0.2576	0.1543	0.1649	0.1257
	img138.bmp	86896	0.875	0.182	0.4008	0.3195	0.3173	0.4600	0.6997	0.2604	0.1496	0.1526	0.1205
	img221.bmp	107856	0.561	0.183	0.3208	0.2506	0.2353	0.3265	0.7032	0.2124	0.0976	0.0907	0.0797
	img211.bmp	143536	0.944	0.190	0.4065	0.3088	0.3534	0.4443	0.7054	0.2771	0.2200	0.2180	0.1701
	img194.bmp	129680	0.863	0.194	0.3736	0.2859	0.3266	0.3914	0.7061	0.2638	0.1724	0.1813	0.1337
	img294.bmp	819952	0.307	0.197	0.3982	0.2308	0.2419	0.3301	0.7060	0.2205	0.1690	0.1630	0.1336
	img208.bmp	132976	0.963	0.202	0.3204	0.2390	0.3543	0.3543	0.7146	0.2790	0.1760	0.1771	0.1527
	img170.bmp	162800	0.984	0.214	0.3752	0.2833	0.3756	0.4257	0.7111	0.3038	0.1991	0.1981	0.1571
	img291.bmp	187328	0.992	0.230	0.3811	0.2940	0.3862	0.4232	0.7195	0.3259	0.2287	0.2392	0.1820
	img288.bmp	84880	0.881	0.259	0.5032	0.3018	0.3725	0.4196	0.7313	0.3301	0.2498	0.2701	0.2352
	img300.bmp	315440	0.519	0.284	0.4701	0.2920	0.3415	0.4278	0.7501	0.2753	0.1622	0.1576	0.1222
	img189.bmp	36592	0.977	0.286	0.4653	0.3830	0.5254	0.5682	0.7425	0.3585	0.2768	0.2882	0.2475
	img303.bmp	450096	0.516	0.293	0.4641	0.3552	0.3851	0.4629	0.7415	0.3027	0.2462	0.2554	0.1961
	img254.bmp	101584	0.816	0.300	0.5529	0.4089	0.4325	0.5696	0.7404	0.3455	0.3004	0.3034	0.2687
	img290.bmp	106752	0.638	0.307	0.4811	0.3652	0.3740	0.5145	0.7508	0.3345	0.2848	0.3087	0.2405
	img295.bmp	217456	0.986	0.327	0.5874	0.5068	0.5079	0.6991	0.7468	0.4120	0.3583	0.3640	0.2778
	img6.bmp	535904	0.910	0.351	0.5686	0.4631	0.4779	0.6143	0.7569	0.3983	0.2668	0.2928	0.2096
	img179.bmp	22576	0.880	0.354	0.5656	0.4403	0.4681	0.5854	0.7711	0.3873	0.2140	0.2403	0.1949
	img190.bmp	268368	0.955	0.386	0.6637	0.5009	0.4925	0.6504	0.7724	0.4175	0.3603	0.3745	0.2943
	img37.bmp	166896	0.848	0.388	0.5882	0.4292	0.4513	0.5440	0.7774	0.3898	0.3575	0.3797	0.3189
	img240.bmp	226064	0.985	0.454	0.8975	0.8057	0.7490	1.0001	0.8340	0.4671	0.0278	0.0312	0.0231
	img39.bmp	201712	0.843	0.461	0.5459	0.4424	0.4999	0.5718	0.8098	0.4278	0.3734	0.3935	0.3533

Corpus	Name	Bits	H(X)	$H_D(X)$	Ratio = (New Size / Original Size)								
					CPRZ	DPRZ	M&EC	HYBC	FRLE	HUFF	LZHF	BZIP	LZMA
Calgary	pic	4105728	0.393	0.153	0.2095	0.1362	0.2133	0.2063	0.6979	0.2080	0.1017	0.0970	0.0853
	geo	819200	0.859	0.924	0.9448	0.8372	0.8590	1.0000	1.1932	0.7130	0.6675	0.5554	0.5177
	obj1	172032	0.930	0.928	1.2539	1.0185	0.9143	1.0001	1.1868	0.7344	0.4795	0.5022	0.4352
	obj2	1974512	0.979	0.953	1.4273	1.1582	0.9894	1.0000	1.2113	0.7591	0.3282	0.3096	0.2498
	paper3	372208	0.996	0.991	1.1810	1.1999	0.8953	1.0001	1.4654	0.5881	0.3879	0.3409	0.3653
	paper2	657592	0.995	0.992	1.1746	1.1984	0.8890	1.0000	1.4557	0.5801	0.3608	0.3051	0.3307
	paper1	425288	0.993	0.993	1.2046	1.2022	0.8915	1.0001	1.4548	0.6206	0.3483	0.3116	0.3238
	book2	4886848	0.994	0.993	1.1923	1.2009	0.8919	1.0000	1.4502	0.5987	0.3374	0.2576	0.2780
	paper4	106288	0.994	0.994	1.1964	1.1964	0.8915	1.0002	1.4570	0.5958	0.4146	0.3905	0.4017
	book1	6150168	0.993	0.994	1.1708	1.1946	0.8802	1.0000	1.4453	0.5720	0.4065	0.3023	0.3395
	progp	395032	0.971	0.995	1.2530	1.0959	0.8723	1.0001	1.4065	0.6122	0.2260	0.2175	0.2089
	paper5	95632	0.990	0.995	1.2298	1.2099	0.8891	1.0003	1.4450	0.6266	0.4158	0.4043	0.4036
	paper6	304840	0.989	0.996	1.2223	1.2046	0.8893	1.0001	1.4406	0.6162	0.3484	0.3229	0.3266
	progc	316888	0.981	0.997	1.2696	1.1862	0.8951	1.0001	1.4404	0.6536	0.3364	0.3165	0.3155
	news	3016872	0.991	0.997	1.2397	1.1991	0.9014	1.0000	1.4549	0.6509	0.3831	0.3145	0.3165
	bib	890088	0.985	0.997	1.2878	1.2354	0.9221	1.0000	1.4804	0.6567	0.3152	0.2462	0.2751
	trans	749560	0.983	0.998	1.2983	1.1656	0.9473	1.0000	1.4705	0.6869	0.2018	0.1913	0.1792
	progl	573168	0.982	0.998	1.2338	1.0868	0.8847	1.0000	1.4462	0.5944	0.2253	0.2174	0.2089
Canterbury	ptt5	4105728	0.393	0.153	0.2095	0.1362	0.2133	0.2063	0.6979	0.2080	0.1017	0.0970	0.0853
	kennedy.xls	8237952	0.636	0.758	0.8537	0.8842	0.5496	0.9508	0.9934	0.4184	0.2010	0.1263	0.0490
	sum	305920	0.875	0.891	1.1744	0.9214	0.8726	0.9934	1.1531	0.6429	0.3356	0.3377	0.2462
	cp.html	196824	0.999	0.993	1.2686	1.2069	0.9116	1.0001	1.4388	0.6619	0.3225	0.3101	0.3100
	grammar.lsp	29768	0.967	0.993	1.2252	1.1356	0.8406	1.0008	1.4311	0.5980	0.3268	0.3451	0.3303
	lcet10.txt	3414032	0.993	0.995	1.1954	1.1710	0.8899	1.0000	1.4613	0.5848	0.3384	0.2518	0.2801
	plrabn12.txt	3854888	0.991	0.996	1.1954	1.2102	0.8840	1.0000	1.4441	0.5743	0.4033	0.3020	0.3431
	xargs.1	33816	0.994	0.996	1.2409	1.2183	0.9106	1.0007	1.4724	0.6291	0.4093	0.4171	0.4135
	fields.c	89200	0.972	0.996	1.2612	1.1558	0.8874	1.0003	1.4536	0.6353	0.2788	0.2726	0.2655
	alice29.txt	1216712	0.986	0.998	1.1975	1.1874	0.8821	1.0000	1.4230	0.5780	0.3561	0.2843	0.3185
	asyoulik.txt	1001432	0.990	0.998	1.2420	1.2138	0.9129	1.0000	1.4568	0.6079	0.3896	0.3164	0.3554

Chapter 5
Summary and Conclusion

5.1 Overall

With the new schemes achieving the primary goal of beating an improved variant of RLE, and nearly matching results of Huffman in image data, the project has successfully completely it's mission. Other schemes that were not planned were developed to improve text-based compression, resulted in achieving some surprising results for images as well.

However, achieving a better compression ratio than Huffman overall is still out of reach for the current variants developed, and Lempel-Ziv is still far off. These two foundational schemes have been tested and still stand as they always have for decades. The developed schemes sole purpose was to break down the walls that so many have built upon these two and has only took a few chips off.

The greatest contribution would be both the Core algorithm and the Rotaries preprocessing. These should prove useful in other aspects and variants of all compression schemes currently developed.

5.2 Future

As the new developed schemes have shown promised, they will be improved upon. Most uniquely in the image compression field, as the single dimensional algorithm nearly matched Lempel-Ziv. Scaling the new algorithms to a multidimensional realm should prove fruitful as preliminary pen-and-paper results as well as current image results have previously shown to reinforce this belief. Another scheme, not mentioned nor empirically tested, will also be developed due to mysterious behavior on pen and paper. This proposed algorithm, if correct, would achieve higher compression ratio than any other algorithm in existence. With a great claim requires great evidence, this task will be pursued with the greatest effort and the greatest scrutiny as there are extreme doubts even to us, the researchers.

Already we have improved the current algorithms mentioned in the results through another trial-by-hand. Overall both the Core and the Dynamic worst cases can be reduced to $\frac{3N}{2}+\log_2 N+C$. The mysterious algorithm is also showing promise for its worst case of $N+C_1\log_2^2 N+C_2$, making it being scrutinized as only Huffman similarly behaves this way, however with the requirement of a table.

References

Claude E. Shannon, Warren Weaver. The Mathematical Theory of Communication. Univ of Illinois Press, 1949. ISBN 0-252-72548-4

US Patent No. 5532693 Adaptive data compression system with systolic string matching logic

A.A. Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons, 1971.

Ziv, Jacob; Lempel, Abraham (May 1977). "A Universal Algorithm for Sequential Data Compression". IEEE Transactions on Information Theory23 (3): 337–343.

Norris, James R. (1998). Markov chains. Cambridge University Press.

Baum, L. E.; Petrie, T. (1966). "Statistical Inference for Probabilistic Functions of Finite State Markov Chains". The Annals of Mathematical Statistics 37 (6): 1554–1563.

Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996, http://www.rfc-editor.org/info/rfc1951

Pavlov I. (2015) LZMA SDK [Software Development Kit] Available from: http://www.7-zip.org/sdk.html

'Apache Commons' (2015) Apache Commons Compress [Software Development Library] Available from: http://commons.apache.org/proper/commons-compress/

History of Lossless Data Compression Algorithms. (n.d.). Available from http://www.ieeeghn.org/wiki/index.php/History_of_Lossless_Data_Compression_Algorithms

Burrows, Michael; Wheeler, David J. (1994), A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation

Marsaglia, George (July 2003). "Xorshift RNGs". Journal of Statistical Software 8 (14).

Appendix B: Agile Programming

Today's technology has caused the commonly traditional planned design being superseded by a more constant update format called the Agile methodology. Coding is done on an iterative basis through simple tasks built on previous accomplishments that further to an obtainable goal. The final product may not be one first envisioned but one that is most suitable and working at the present time and technology. Something that was designed and planned for two years before implementation may have no relevance for today's technological advances. The more tentative and iterative approach Agile brings, gives the user and programmer combined control in the design process where both sides can see a clear, reasonable, and obtainable goal. In addition, even before the final product is finished, iterations that come before may be partially suitable for increase in work productivity. This benefit alone makes it an attractive methodology of design.

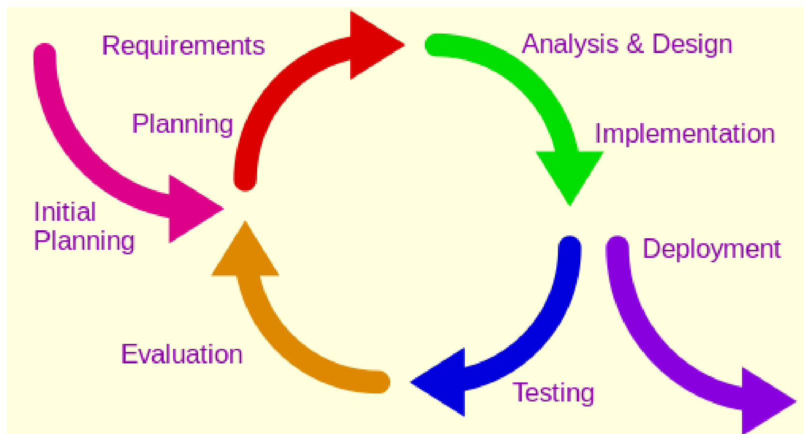


Fig 2.3.1 Iterative development flow model

Theoretical concepts will be put through this method, where each derivative of the new scheme and its algorithm will be constructed, kept, and tested. The first task is implementing the core alone, then dynamic, while implementing the rotary algorithm to both.