Assessing LZW and Huffman Coding for DNA Sequence Data Compression

Miro Manuel L. Hernandez
Electronics and Computer Engineering Department
De La Salle University - Manila
Manila, Philippines
miro_hernandez@dlsu.edu.ph

Matthew R. Orga
Electronics and Computer Engineering Department
De La Salle University - Manila
Manila, Philippines
matthew_orga@dlsu.edu.ph

Dominic P. Yu
Electronics and Computer Engineering Department
De La Salle University - Manila
Manila, Philippines
dominic_yu@dlsu.edu.ph

Abstract— The Lempel Ziv-Welch (LZW) and Huffman Coding algorithms are widely used for lossless data compression in various applications. However, this project focuses on the comparison on which algorithm would perform better when compressing DNA sequence data. The project is limited to a DNA sequence of 100 letters for testing, including a manual computation, and implementation programs that will be simulated in the MATLAB IDE.

Keywords— algorithm, DNA, Huffman Coding, LZW, MATLAB IDE

I. Background of the Study

The Lempel Ziv-Welch (LZW) algorithm is a lossless data compression algorithm that has been widely used in various applications due to its high compression ratio and efficiency. It was first introduced by Lempel and Ziv in 1977 and later improved by Welch in 1984. LZW has been implemented in many standard software programs and is still commonly used today in applications such as GIF images, PDF files, and UNIX compress utility. Despite its widespread use, the implementation and optimization of the LZW algorithm have been a subject of research interest in recent years. Several studies have investigated the performance of the LZW algorithm in different scenarios and proposed various optimization techniques[1],[2],[3],[4].

The Huffman Compression Algorithm is a widely used data compression method that is particularly suited for compressing text files. The algorithm works by analyzing the frequency of occurrence of each character in the input file and assigning shorter codes to frequently occurring characters and longer codes to less frequently occurring characters [5]. Advantages of this approach allows for significant compression of the input file, as the most frequently occurring characters are given the shortest codes, reducing the number of bits required. The Huffman Compression Algorithm has been widely studied and applied in various fields, including computer science, information theory, and signal processing. [6].

Both the LZW and the Huffman compression algorithms have been studied in relation to the problem of DNA sequence data compression. In a recent study, They proposed the use of LZW algorithm to compress DNA sequence data using indexed multiple dictionaries [7]. Meanwhile, Al-Okaily and colleagues went into the path of using Huffman as a means of compressing DNA Data [9].

II. Problem Statement

Biological sequence data, which is the result of DNA sequencing experiments, is an essential data set under analysis in the current era of various research communities. It comprises the four bases Adenine, Guanine, Cytosine, and Thymine, forming the building blocks of cells in every organism [7]. For context, these biology laboratories generate large amounts of information (currently, in Petabytes or 250), consisting of genomic information, proteins, and small molecules. Further, the data generated increases approximately two-fold every year [10]. The growth of DNA sequence data presents challenges to data storage and management. Repositories like GenBank, DDBJ, and EMBL provide sequence data and basic annotation, but storage of raw data from new technologies such as Illumina GAII or AB SOLiD is problematic due to large image sizes, with storage costs exceeding data generation costs. Additionally, redundant data generated by new sequencing technologies adds to the challenge of data storage and management. As reference genome sequences become more available, primary sequence repositories are expected to be replaced by smaller sequence variation databases that provide a more valuable user experience. Better and more efficient data storage solutions are needed to address these challenges. [11].

III. Objectives

The research aims to aid in the data problem within the scientific field, more specifically, Biology. To attain this, the researcher will implement and evaluate the performance of different compression algorithms namely;

(1) LZW Algorithm
(2) Huffman Code

The subject or the main data to be compressed is genomic information (DNA Sequence), specifically the DNA Characters A, G, C, and T, which are Adenine, Guanine, Cytosine, and Thymine respectively.

IV. Methodology

The project simulation was tested with the MATLAB IDE. Both of the compression methods, namely the LZW algorithm and the Huffman Code algorithm were simulated in the said IDE and the output values and compression ratio of the two compression algorithms will be compared in the results and discussions section of this project. The sample

DNA sequence that was used in both compression methods is:

GTAGCGCTAAAAGTCCATAGCACGTGCATCCCAACG
TGGCGTGCGTACAGCTTGACCACCGCTTCACGCTAA
GGTGCTGGCCACATGCTAAATTGATGCG

The DNA sequence contains 100 total letters, and to apply the the lessons discussed from the lecture meetings, a manual computation for the input sequence will be included in the results and discussions part of this project.

For LZW, the researchers will experiment with two different index entries. The first one will be tailored to DNA Sequencing while the second one will be making use of the base LZW indexing that accounts for other ASCII characters. To simplify, this means that the first indexing will start from 5, since there are only four characters needed in DNA sequencing while the second indexing will start at 256.

For the specialized indexing the values will be:

Index	Entry
1	A
2	C
3	G
4	T

Table 1. Specialized LZW Indexing for DNA Sequence

Index	Output	Entry
5	3	GT
6	4	TA
7	1	AG
8	3	GC
9	2	CG
10	8	GCT
11	6	TAA
12	1	AA
13	12	AAG
14	5	GTC
15	2	CC
16	2	CA
17	1	AT
18	6	TAG
19	8	GCA
20	1	AC
21	9	CGT
22	4	TG
23	19	GCAT
24	4	TC
25	15	CCC
26	16	CAA
27	20	ACG
28	5	GTG
29	3	GG
30	8	GCG
31	28	GTGC
32	21	CGTA
33	20	ACA
34	7	AGC
35	2	CT
36	4	TT
37	22	TGA
38	20	ACC
39	16	CAC
40	15	CCG
41	10	GCTT
42	24	TCA
43	34	ACGC
44	35	CTA
45	13	AAGG
46	31	GTGCT
47	22	TGG
48	8	GCC
49	39	CACA

50	17	ATG
51	19	GCTA
52	12	AAA
53	17	ATT
54	37	TGAT
55	22	TGC
56	9	CG-

Table 2. LZW Algorithm Manual Computation with Specialized Indexes

```
"GT"    ↦ 3
"TA"    ↦ 4
"AG"    ↦ 1
"GC"    ↦ 3
"CG"    ↦ 2
"GCT"   ↦ 8
"TAA"   ↦ 6
"AA"    ↦ 1
"AAG"   ↦ 12
"GTC"   ↦ 5
"CC"    ↦ 2
"CA"    ↦ 2
"AT"    ↦ 1
"TAG"   ↦ 6
"GCA"   ↦ 8
"AC"    ↦ 1
"CGT"   ↦ 9
"TG"    ↦ 4
"GCAT"  ↦ 19
"TC"    ↦ 4
"CCC"   ↦ 15
"CAA"   ↦ 16
"ACG"   ↦ 20
"GTG"   ↦ 5
"GG"    ↦ 3
"GCG"   ↦ 8
"GTGC"  ↦ 28
"CGTA"  ↦ 21
"ACA"   ↦ 20
"AGC"   ↦ 7
"CT"    ↦ 2
"TT"    ↦ 4
"TGA"   ↦ 22
"ACC"   ↦ 20
"CAC"   ↦ 16
"CCG"   ↦ 15
"GCTT"  ↦ 10
"TCA"   ↦ 24
"ACGC"  ↦ 27
"CTA"   ↦ 35
"AAGG"  ↦ 13
"GTGCT" ↦ 31
"TGG"   ↦ 22
"GCC"   ↦ 8
"CACA"  ↦ 39
"ATG"   ↦ 17
"GCTA"  ↦ 10
"AAA"   ↦ 12
"ATT"   ↦ 17
"TGAT"  ↦ 37
"TGC"   ↦ 22
```

Figure 1. LZW Algorithm Specialized Indexes Output

Index	Entry
65	A
67	C
71	G
84	T

Table 3. Initial ASCII codes to be encoded

For second indexing of LZW the values will be:

Index	Output	Entry
256	71	GT
257	84	TA
258	65	AG
259	71	GC
260	67	CG
261	259	GCT
262	257	TAA
263	65	AA
264	263	AAG
265	256	GTC
266	67	CC
267	67	CA
268	65	AT
269	257	TAG
270	259	GCA
271	65	AC
272	260	CGT
273	84	TG
274	270	GCAT
275	84	TC
276	266	CCC
277	267	CAA
278	271	ACG

279	256	GTG
280	71	GG
281	259	GCG
282	279	GTGC
283	272	CGTA
284	271	ACA
285	258	AGC
286	67	CT
287	84	TT
288	273	TGA
289	271	ACC
290	267	CAC
291	266	CCG
292	261	GCTT
293	275	TCA
294	278	ACGC
295	286	CTA
296	264	AAGG
297	282	GTGCT
298	273	TGG
299	259	GCC
300	290	CACA
301	268	ATG
302	261	GCTA
303	263	AAA
304	268	ATT
305	288	TGAT
306	273	TGC
307	260	CG-

Table 4. LZW Algorithm Manual Computation

Link for the LZW Manual Computation Image (Solution):
https://drive.google.com/file/d/1PJICUWAZg-ogcsqSHoMhuRyvWUNLi0G-/view?usp=share_link

V. Simulation Results and Discussions

LZW Algorithm MATLAB Implementation:

The MATLAB LZW Algorithm coding program made by the researchers can be accessed through this link:
https://github.com/dominicyu704/LZW_MATLAB

```
D2 =

  dictionary (string ⟼ double) with 51 entries:

    "GT"     ⟼ 71
    "TA"     ⟼ 84
    "AG"     ⟼ 65
    "GC"     ⟼ 71
    "CG"     ⟼ 67
    "GCT"    ⟼ 259
    "TAA"    ⟼ 257
    "AA"     ⟼ 65
    "AAG"    ⟼ 263
    "GTC"    ⟼ 256
    "CC"     ⟼ 67
    "CA"     ⟼ 67
    "AT"     ⟼ 65
    "TAG"    ⟼ 257
    "GCA"    ⟼ 259
    "AC"     ⟼ 65
    "CGT"    ⟼ 260
    "TG"     ⟼ 84
    "GCAT"   ⟼ 270
    "TC"     ⟼ 84
    "CCC"    ⟼ 266
    "CAA"    ⟼ 267
    "ACG"    ⟼ 271
    "GTG"    ⟼ 256
    "GG"     ⟼ 71
    "GCG"    ⟼ 259
    "GTGC"   ⟼ 279
    "CGTA"   ⟼ 272
    "ACA"    ⟼ 271
    "AGC"    ⟼ 258
    "CT"     ⟼ 67
    "TT"     ⟼ 84
    "TGA"    ⟼ 273
    "ACC"    ⟼ 271
    "CAC"    ⟼ 267
    "CCG"    ⟼ 266
    "GCTT"   ⟼ 261
    "TCA"    ⟼ 275
    "ACGC"   ⟼ 278
    "CTA"    ⟼ 286
    "AAGG"   ⟼ 264
    "GTGCT"  ⟼ 282
    "TGG"    ⟼ 273
    "GCC"    ⟼ 259
    "CACA"   ⟼ 290
    "ATG"    ⟼ 268
    "GCTA"   ⟼ 261
    "AAA"    ⟼ 263
    "ATT"    ⟼ 268
    "TGAT"   ⟼ 288
    "TGC"    ⟼ 273
```

Figure 2. LZW Algorithm MATLAB Output

Based on the results obtained, the manual computation and the MATLAB implementation produced the same results.

The only difference is that in the manual computation, the last index is 307, containing an output of 260, and the entry would be CG- if the sequence is theoretically continuous. If the index 307 is not included, then both implementations yield the same result. However, the compression ratio will then be $\frac{100}{51}$ or 1.9608. If the index 307 is included, the compression ratio will be $\frac{100}{52}$ or 1.9231. For the LZW algorithm, the higher the compression ratio, the better since it compresses the data more effectively. However, it does not mean that the LZW algorithm is always the best compression ratio for all cases, there are still other considerations such as compression speed, decompression speed, and the time complexity of the compression algorithm.

Huffman Coding Manual Computation:

Source Code	Symbol
01	A
10	C
11	G
00	T

Table 5. Source Code from Manual Computation

Source Code	Symbol
10	A
00	C
01	G
11	T

Table 6. Source Code from Manual Computation

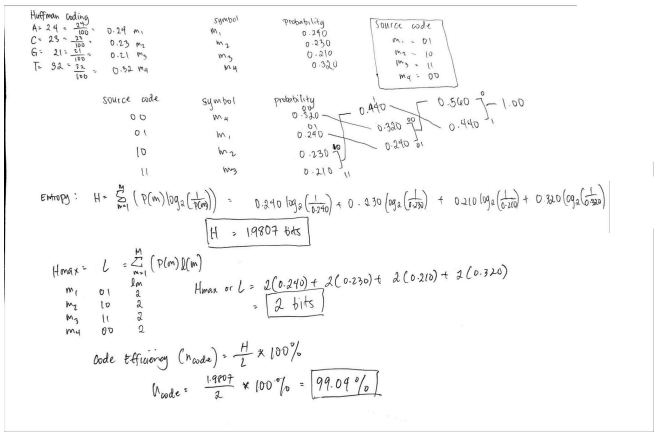

Figure 5. Huffman Code Manual Computation

Link of the image for better viewing:
https://drive.google.com/file/d/1FM4wrqPR9gKSkGG0_aL_YVq3dUet2O4N/view?usp=share_link

Huffman Coding Source Code Output from MATLAB:

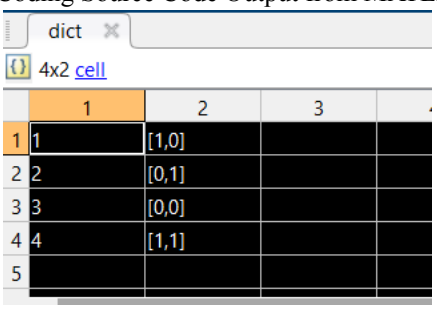

Figure 3. Source Code generated from MATLAB IDE

```
symbols = ["A", "G", "C", "T"];
numeric_value = [1, 2 , 3 , 4];
```

Figure 4. Mapping Values generated from MATLAB IDE

The MATLAB Huffman coding program made by the researchers can be accessed through this link:
https://github.com/meero30/Huffman_coding_MATLAB?fbclid=IwAR3-RDMUgm1jYbzwR0Pgkcn-oexLs-WyCVnbil60lptmSCHoS7zi35N4Q8w

The only difference from the output of the MATLAB program is that the Huffman Coding dictionary addon uses maximum variance, whereas the manual computation uses minimum variance [13]. With the MATLAB code, the source code that is generated is just the opposite bits. For example, the source code for the input *A* is 1,0 from the MATLAB program, but is 0,1 from the manual computation. From the manual computation, the probability values will be found by getting the amount of the total appearances of each letter in the sequence. Since the total letter count in the DNA sequence is determined to be 100, the times the letter appeared should be divided to 100 and the result will be the probability of the symbol.

maximum	39
bit_count_code	312

Table 7. Bit count code for specialized indexes LZW algorithm

bit_count_code	468

Table 8. Bit count of LZW output

bit_count_compressed	200

Table 9. Bit count of Huffman Coding output

VI. Analysis and Conclusion

The results of the LZW MATLAB program are equivalent to the manual computation. Meanwhile, the Huffman Coding MATLAB Program has different source code values when compared to the manual computation. This difference is the result of two different algorithms of getting the Huffman Source code. The MATLAB Program used maximum variance algorithm while the manual computation used minimum variance algorithm. Albeit the difference, both of the source code lengths are equal, hence both will have the same coding efficiency. The main focus of this project is to compare which compression algorithm would perform better when it comes to the compression of DNA sequence data. Based on the results, both compression

algorithms resulted in a high compression ratio. The LZW algorithm has a compression ratio of 1.9608, and the Huffman Coding algorithm has a code efficiency of 99.04%. With the results, both algorithms are capable of achieving high levels of compression. However, Huffman Coding could become more efficient when it comes to DNA sequence data compression since it can assign shorter codes to frequently occurring nucleotides and longer codes to the rarely occurring nucleotides. It can be concluded that using the Huffman Coding for DNA sequence data compression would be much more efficient since it can reach a high compression ratio while still maintaining a high code efficiency.

Comparing the bit count of the LZW and Huffman code compressed, it would seem that Huffman is more efficient when it comes to compressing DNA sequences due to it having almost half the bitcount of the LZW output. The reason for this is due to the Huffman code only needing 2 bits to represent 1 output character. Meanwhile, in the context of the example DNA Sequence, LZW needs 9 bits to represent 1 output character. The reason behind this is due to the immediate high index of LZW, which will start at 256. A higher index means needing a higher number of bits to represent the number. The specialized variant of LZW fared better, having less bit count against its base variant but not enough to beat Huffman coding.

To conclude, in a scope of 100 characters of DNA sequencing, Huffman Coding is the better choice compared to LZW, regardless of the indexing of the LZW.

Further avenues of studies would be to test a much larger *N* size of DNA Sequence characters, this due to a possible hypothesis of LZW being more effective in larger quantities of data.

References

[1] Welch, T. A. (1984). A technique for high-performance data compression. IEEE Computer, 17(6), 8-19.

[2] Lempel, A., & Ziv, J. (1977). A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3), 337-343.

[3] Storer, J. A., & Szymanski, T. G. (1982). Data compression via textual substitution. Journal of the ACM, 29(4), 928-951.

[4] Navarro, G., & Baeza-Yates, R. (2001). Compressed data structures: A survey. ACM Computing Surveys (CSUR), 33(4), 427-469.

[5] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.

[6] Sayood, K. (2017). Introduction to data compression. Academic Press.

[7] Keerthy and Priya (2017). Lempel-Ziv-Welch Compression of DNA Sequence Data with Indexed

[8] Multiple Dictionaries. Retrieved from https://www.ripublication.com/ijaer17/ijaerv12n16_26.pdf

[9] Al-Okaily et al. (2017). Toward a Better Compression for DNA Sequences Using Huffman Encoding. Journal of computational biology : a journal of computational molecular cell biology, 24(4), 280–288. https://doi.org/10.1089/cmb.2016.0151.

[10] Marx V. (2013). The Big Challenges of Big Data. Retrieved from: https://www.nature.com/articles/498255a

[11] Batley, J., & Edwards, D. (2009). Genome sequence data: management, storage, and visualization. Biotechniques, 46(5), 333-336

[12] IBM. (2023). ASCII, decimal, hexadecimal, octal, and binary conversion table. Retrieved from https://www.ibm.com/docs/en/aix/7.2?topic=adapters-ascii-decimal-hexadecimal-octal-binary-conversion-table

[13] Mathworks (n.d.). huffmandict. Retrieved from https://www.mathworks.com/help/comm/ref/huffmandict.html?fbclid=IwAR2VDFCjEAOqqsB_UU26kCQ2MlNTTrkYsPuQmBk_uCujW1kIjxgcPmOd0L0