Fast Canonical Huffman Decoder

*

Sabin Belu
Doctoral School ETTI
Univ. Politehnica of Bucharest
Bucharest, Romania
bsabin2012@gmail.com

Daniela Coltuc
Faculty of Electronics, Tc. and Information Technology,
Univ. Politehnica of Bucharest
Bucharest, Romania
daniela.coltuc@upb.ro

***Abstract*—We propose a novel method to decode canonical Huffman codes specifically designed to decode multiple symbols in one decoding cycle. The encoding method used is the classic canonical Huffman encoding. Our decoding method is specifically designed for speed and provides big throughput of decoding data; within our tests, we have achieved a decompression output of more than 2.1GiB/s when decompressing highly redundant data. Our method holds minimal storage requirements for decoding tables while still operating at very high speeds. We will also show that for limited-length canonical $Huffman$ codewords that extend up to 12 bits, the memory size and construction times of the decoding tables are negligible.**

***Index Terms*—Canonical Huffman decoding, Fast decoding, Entropy Coder.**

I. Introduction

Huffman codes have been around since 1951 when they were invented by D.A.Huffman [1]. Huffman strived to find the most efficient method of representing a symbol to be encoded into a more compact form. He solved this problem with a simple idea, an idea that brought him a highest Information Theory rank of underpinning technical achievers of all times. Donald E. Knuth, the author of the multi-volume series "The Art of Computer Programming" [2] wrote, *"Huffman code is one of the fundamental ideas that people in computer science and data communications are using all the time"*.

The Huffman method is not a singularity when it comes to producing compact prefix codes representation of symbols from an alphabet. Robert Fano [3], who was at that time D.A.Huffman's college professor, is also known to be the co-author of a less optimal method of entropy encoding. The method is called Shannon-Fano after its inventors, Claude Shannon [4] and Robert Fano.

In theory, Huffman codes approach optimality but when it comes to implementation, the method is much less optimal especially when fewer symbols are encoded. We will explain this in Section III.

New progressive codes and coding method had to be found in order to optimally solve this problem and as a result a new set of Huffman codes that follow certain rules have been lately proposed.

Such codes are called the Canonic Huffman codes [5], and they possess unique properties that will allow us to solve step 1 and 2, the transmission/ storage and retrieval of the information required for reconstruction of the input symbols in an optimal way. Such properties define the canonical Huffman codes to be sequential. They will be discussed in more details during this article, but it is worth mentioning that the most important property of such codes has dramatically changed the paradigm of Huffman coding. This code property, called *consecutive value property*, allows most of the canonical Huffman codes to be automatically generated in sequential order when the number of bits per codeword is provided.

There is no longer necessary to store the Huffman tree to reconstruct it before the decoding process and pick the left sub-node or the right sub-node of the current node based on the input bit. Given this consecutive value property, which can be seen as an *auto generation* of sequential codewords, a list of incremented codes could be gathered in a table which dramatically reduces decoding time.

Therefore, we propose herein a new decoding method based on the canonical codes consecutive value property and also on the notion of instantaneous codes, that will be further described in details. We have designed the Fast Canonical Huffman Decoder (FCHD) with speed in mind and with the intent of minimizing the extra steps taken to decode symbols or portions of input, completely guided by the KISS [1] principle. The compiled code reaches very high speed during our testing and outperformed all the known Huffman canonical implementations. Tests are conducted on the same test machine, under the same test conditions.

II. Related Work

Sieminsky proposed in [6] a solution in which he avoids manipulation of individual bits when decoding Huffman codewords but it has increased memory requirements: for an arbitrary number of bits, the algorithm uses multiple decoding tables, which are chosen based of the current context. For instance, the algorithm could interpret '011' as the suffix for a codeword plus a codeword for another symbol. It could also

[1] KISS, an acronym for Keep it Simple, Stupid, is a design principle noted by the U.S. Navy in 1960.

be the codeword for two symbols. It cannot be the prefix for another codeword, since these are prefix codes. The decoding interpretation varies according to its previous group of bits called decoding context. The decoding context, for example 101, will determine the decoding table while the decoding group selects an appropriate entry in the first table. The size of the decoding tables grows at a very fast rate when the number of the different symbols is growing as well, because the number of entries is given by the number of possible values of a certain number of bits multiplied by all the possible numbers. We believe that this method does not qualify as a fast decoding method since its memory requirements make it unfeasible to fit into modern computers internal cache sizes, which could very well store between 256KiB to 2MiB of data these days.

In [7], Tanaka proposed a semi-autonomous finite-state sequential machine for decoding Huffman codes. The Huffman tree structure is presented as a 2D array, which is used to decode Huffman codes as a state transition table of this finite-state decoding automaton. Since this may look as an advantage, we consider that the algorithm provides more steps than needed to decode a single Huffman symbol.

Renato describes in [8] a node-transition algorithm implemented using fast prefix code decoding tables. This improves decoding performance at the expense of memory usage. For a sequence of bits of fixed size, all possible leaf nodes can be pre-computed forehand and stored in a table. Renato states that for a fixed codeword length of k bits, $2k$ possible node transitions or bit sequences must be considered at every node. This allows the decoder to jump effectively for any node to another in the tree code by processing bits simultaneously instead of single bits. Although there are some advantages by processing multiple bits, the biggest disadvantage that is obvious in this method is that the algorithm requires 20 node transition storage entries only for $k = 2bits$, since it also includes the ROOT as stated by Renato: *"leaf nodes have all the same node transition tables as the root"*. For $k >= 8bits$, the table storage and memory requirements could reach $O(256 \times 28 \times 8) \times sizeof(transition\_node)$, which we note to be prohibitive.

The node transitions and symbol output tables do not get stored in the archive, since they can be reconstructed in a pre-processing step before the decoding. However, if we consider the extra time spent before the decompression steps to build more than 64K transitions when dealing with only 8-bit states to be prohibitive, we can conclude that this greatly reduces every savings this algorithm could achieve in decoding time.

Alistair Moffat and Andre Turpin propose in [9] two canonical Huffman decode methods: algorithm ONE-SHIFT and algorithm TABLE-LOOKUP. Similar to the algorithms previously presented, these two new algorithms process only one symbol at each decoding step. They involve a linear search before the table lookup operation, performed in order to establish the length of the codeword associated with a certain base value, coded as a incremental step and condition for the value associated with the length.

In [10], Hirschberg and Debra Lelewer present Method A (A1 and A2) and Method B, with its variants B1 and B2 for Canonic Huffman Codewords Decompression. They present a particular kind of Huffman tree, which when constructed processes multiple symbols in a sequence called $string$. In this initial phase, the algorithm compresses the input source by replacing strings of symbols with pointers to a dictionary. The formed dictionary is a collection of N strings of variable bit-lengths. In the decoding phase, one Huffman codeword will decode a look-up index with respect to this dictionary, which has to be re-constructed during the decompression phase, adding extra time and memory resources.

In a nut shell, Huffman decoding algorithms presented in [10] do not work with symbols, but with a collection of strings, pre-parsed with an initial overhead into a dictionary. Therefore, if we relate only to Huffman decoding, their algorithm does not decode multiple dictionary symbols, or in this case, dictionary indexes. It still operates at a one symbol level. Memory requirements: Method A1 requires $3 * n * 2$ bytes and Method B1 $2*n+54$ bytes in a typical application. Method B1 seems to have greater storage requirements than Method A2. In addition to the time required to receive the data, the decoder performs $O(n)$ operations in setting up the address table and $O(L)$ operations in constructing limit and base tables. This is on top of the operations related to the initial phase dictionary, which has to be recreated, with extra time and space requirements.

Unlike any of the classic implementations that we have surveyed, our FCHD implements an algorithm that allows very fast decoding of multiple symbols in one decoding cycle.

III. Fast Canonical Huffman Decoder

When performing symbols encoding, Huffman method consists of the following steps:

1) Count the amount of times each $symbol$ appears in the input stream that is, assign them probabilities.
2) Create a binary tree, allocate two sub-nodes for each internal node and bind each leaf node to an individual symbol depending on its frequency (or probability).
3) Assign codewords to symbols located in leaf nodes by using a set-notation when traversing the tree from the root to the leaves: assign bit 0 to the left sub-node and bit 1 to the right sub-node for each encountered node.
4) Create the output stream by replacing the input symbols with their respective codewords.

In order to perform decoding of encoded symbols, Huffman steps are the following:

1) The input symbol frequencies or assigned probabilities must be recovered from a transmission channel or read from another source.
2) The Huffman binary tree is constructed following the same steps used during the encoding process.
3) Decompose the symbol codeword into bits and start traversing the tree, following the exact set-notation as for the encoding process: bit 0 for left traversal bit 1 for right traversal.

4) When a leaf node is reached, record the associated symbol and continue decoding the output stream from step 3, until the output stream is depleted.

Canonical Huffman is meant to correct two major deficiencies of classic Huffman coding [1].

- *Huffman tree storing and transmission*, an expensive step in terms of storage requirements and transmission even for a small set of input symbols.
- *Traversing the Huffman tree* to perform decoding.

The classic Huffman tree traversal, choosing left of right child node depending on the current bit value, is extremely exhausting in terms of decompression speed ; such traversal only advances 1 bit per cycle or iteration while decoding only one symbol.

Huffman coding and in particular, the decoding part, uses the same information to reconstruct the original input stream. In practice, Huffman is far from optimality, because of step 1 i.e., storing or transmission of tree information over an information channel. Huffman's method also proves to be less scalable for larger alphabets, and apart from the set-notation of the left node to 0 and right node to 1, the classical Huffman tree and generated codes have no additional properties we can rely upon for optimal storage or transmission.

A. The Canonical Huffman Encoding Algorithm

Let s_1, s_2, ..., s_N be the following source alphabet symbols, sorted according to their frequency of appearance in the input steam. s_1 being the most frequent symbol and s_N being the least frequent. $l_{s_1} \leqslant l_{s_2} \leqslant \ldots \leqslant l_{s_N}$ the $bit-lengths$ of the corresponding $codewords$ generated by the classic Huffman tree.

The pseudo-code for the Canonical Huffman coding could be described as following:

1) $codeword(s_1) = 0$
2) $codeword(s_N) = (codeword(s_{N-1}) + 1) \ll (l_{s_N} - l_{s_{N-1}})$ [1]

The algorithm initializes $codeword$ for symbol s_1 with 0 in step 1. It then iterates through all the input symbols and generates the next codeword by using the second step formula.

We give as example the input stream with the symbol frequencies $Freq(A) = 9$, $Freq(T) = 5$, $Freq(C) = 3$, and $Freq(G) = 1$.

It is known that symbol correlation is ignored by an order-0 entropy coder like Huffman. We chose a symbol-correlated example to demonstrate our FCHD could achieve N-symbols/cycle decompression when working on highly redundant files, such as 24bit bitmaps.

We note $Freq[]$ the frequency table and $Freq[s]$ the amount of times a symbol s appears in the input stream; after counting the number of appearances in the input stream, e.g. s appears for p times in the input stream, the value is recorded in the frequencies table at position indexed by the ASCII[1] value of symbol s, $Freq[s] = p$.

We proceed with the construction of the Huffman tree by using the symbol frequencies alone.The $codewords$ and their $bit-lengths$ are given in Table I.

TABLE I
SYMBOL FREQUENCIES WITH ASSOCIATED HUFFMAN AND CANONICAL CODEWORDS

Index	Symbol	$Freq_{10}$	Codeword Length	$Canonic\ Code_2$	$Huffman\ Code_2$
0	A	9	1	0	1
1	T	5	2	10	01
2	C	3	3	110	000
3	G	1	3	111	001

The canonical rules from above, 1st and 2nd steps, are used to generate the codewords by using only codewords bit-lengths:

- **Step 1** $codeword(A) = 0$
- **Step 2** $codeword(T) = (0 + 1) \ll (2 - 1) = 10_2$
- **Step 3** $codeword(C) = (2 + 1) \ll (3 - 2) = 110_2$
- **Step 4** $codeword(G) = (6 + 1) \ll (3 - 3) = 111_2$

The canonical Huffman codes are shown in Table I. A series of properties that make these codes very suitable for fast decoding have been revealed since their conception. Among them, one very remarkable property is the *Consecutive-values* property.

For any given bit length $l > 1$, the associated codewords $C_1(l), C_2(l) \ldots C_N(l)$ represent a consecutive range of positive integers:

$$C_1(l) < C_2(l) < ...C_N(l), \tag{1}$$

Another canonic Huffman property is *Half–of-Successor* property: given any $k < l_{max}$, (l_{max} is the maximum codewords bitlength), we note $u = C(k)$ the smallest $codeword$ of length k and $v = C(k + 1)$ the largest codeword of bitlength $k + 1$. The following relationship exists between u and v:

$$u = \left(\frac{v+1}{2}\right) \tag{2}$$

B. Construction of FCHD decoding table

We propose a fast multi-symbol canonical Huffman decoding solution, implemented using a single lookup table. This lookup table requires minimum time for construction and reduced storage in the output file. It will be further referred to as the FCHD decoding table. We place a series of input symbols in the decoding table, $s_p \ldots s_k$, at a pre-calculated index in this table. We show how these indexes are constructed below.

In order to construct the decoding table, we partition the Huffman binary output stream into segments that follow two rules:

[1]Increment the code for symbol s_{n-1} by 1 and perform shift left operation with $shift\_bits$ equal to $l_{s_n} - l_{s_{n-1}}$.

[1]ASCII (/æski/ ASS-kee), abbreviated from American Standard Code for Information Interchange, is a character encoding standard for electronic communication.

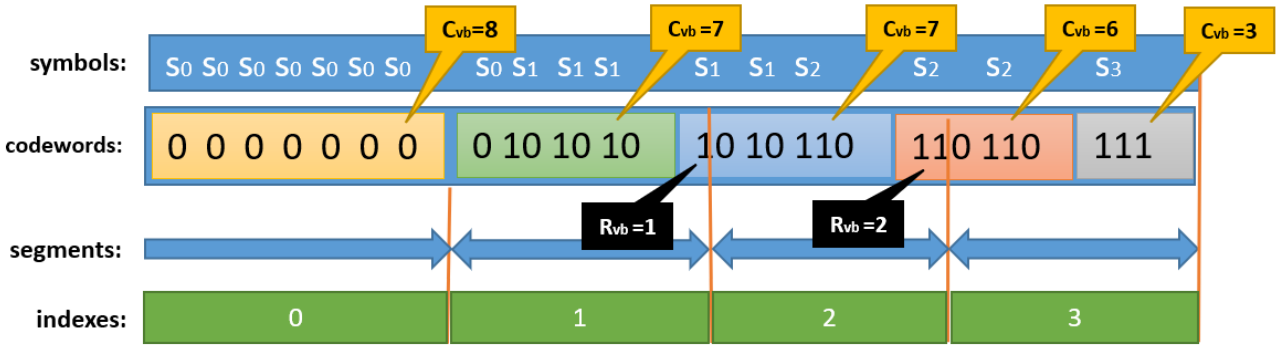

Fig. 1. Creation of decoding segments and indexes.

A. **The greedy rule**: Each segment tries to maximize the number of instantaneously decodable codewords it contains.

B. **The size-N rule**: The *cumulated codeword length* is less or equal to an arbitrary chosen N. N defines the decoding table size and gives a new name to our FCHD, the $N-bit\ FCHD$.

For simplicity, we chose to present this algorithm for $N = 8$. Let the binary stream in Fig. 1 be the result of encoding the message $s_0, s_0 \ldots s_2, s_3$ with the canonical Huffman codes from Table I. The FCHD decoding table is built on the fly as follows:

1) Parse the coded stream in segments. A segment is a set of codewords $c_i \ldots c_k$ such that their cumulated bit-lengths $l_{c_i} \ldots l_{c_k}$ do not exceed N:

$$C_{vb} = \sum_{j=i}^{k} l_{c_j} \leq N \tag{3}$$

See symbols and codewords in Fig. 1.

2) Calculate decoding indexes from C_{vb} and the remainder R_{vb}, if any remainder. The remainder R_{vb} exists only if the cumulated codeword length $C_{vb} > N$, and has the following value:

$$R_{vb} = C_{vb} - N \tag{4}$$

See segments, $R_{vb} = 1$, $R_{vb} = 2$ in Fig. 1.

3) Using FCHD table index, we then store C_b and R_b at the location of the decoding table where the decoding index points to. The remainder is a leftover from the next instantaneously decodable codeword c_{k+1} which is no longer processed in Step 1, due to rule A.

A $n-bit\ FCHD$ is an implementation of our algorithm which deploys a decoding table size 2^n e.g., the $8-bit\ FCHD$ decoding table contains 256 elements.

IV. Experimental results

We have conducted a series of experiments on various data types. The main purpose of these tests is to evaluate the decoding speed for our Canonical Huffman decoding algorithm and to establish how it compares with other entropy coders.

We use the following generic data sets which we believe to be suitable for an order-0 entropy coder:

- **Test data 1)**: compressible nucleotide files with four symbols alphabet $\{A, C, G, T\}$. These test files contain RNA reference genome sequences for various viruses & variations of such, downloaded from the National Library of Medicine, Bethesda, MD, USA [17]
- **Test data 2)**: highly compressible files with only one symbol being more frequent. These are 24bit-bitmaps where the most frequent symbol comes from high repetitions of the white color from the bitmap background.
- **Test data 3)**: ASCII files with uniform distributions of symbols. This is a mixture of text source files, binary temp files, and compiled executable data.
- **Test data files 4) through 7)**: artificial files containing tokens or markers from LZFG–A1 [18] output file. LZFG-A1 method encodes $< literals >$ and $< length, distance >$ pairs into tokens and markers called $LitRuns$ an $LenRuns$. Each of our test files 4) through 7) contains one type of these LZFG-A1 markers or tokens.
- **Test data files 8) through 12)**: artificially generated files

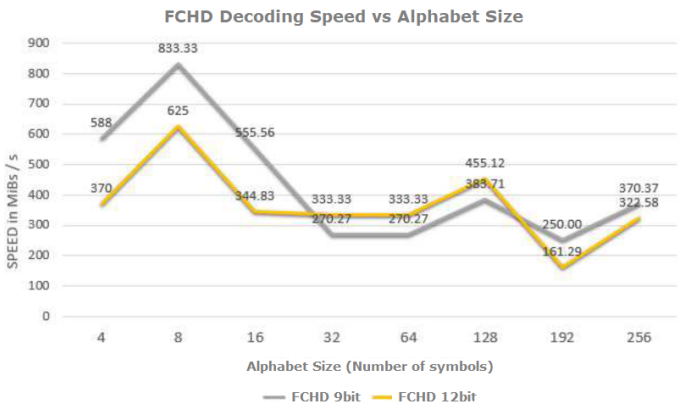

Fig. 2. Decoding speed of N-bit FCHD variations vs alphabet size.

TABLE II
FCHD DECODING SPEED (MIB/S).

Test Data	9-bit FCHD	12-bit FCHD	Huff-0	Huff-X	Range-0	Zlib Huffman
1	966.93	**985.71**	162.51	169.55	60.46	110.24
2	400.53	**501.81**	182.64	114.36	59.24	83.42
3	344.83	**385.41**	127.21	128.93	76.44	69.16
4	**230.11**	196.11	124.66	131.37	54.34	69.61
5	389.69	**626.71**	140.51	142.42	67.42	75.29
6	299.65	**370.51**	129.72	133.91	54.11	74.11
7	383.71	**455.10**	137.6	134.8	108.11	70.62
8	**833.33**	625.01	172.00	173.10	60.60	90.91
9	**555.56**	500.83	173.10	173.10	61.60	126.58
10	270.27	**333.33**	170.60	169.30	58.90	106.38
11	270.27	**322.58**	165.50	165.5	58.90	70.62
12	**383.71**	169.51	137.60	134.80	108.10	70.62

that contain a fixed number of different symbols with random probabilities within the file. Data was generated with the uniform generator function rand(), using modulo N, where N is the alphabet size and ranges from 1 to 4.

We compare our FCHD implementation against the following publicly available entropy coders:

- **Huff-0** [14] is developed by FB developer, Y. Collet. It is an order-0 Huffman entropy coder using canonical Huffman codes [5]. This decoder is currently used in large-scale applications like ZHuff [15], precursors of Zstandard [16].
- **Range-0** [19] is an order-0 range coder[4] also developed by Y. Collet.
- **Huff-X** [20] is a mixed entropy coder with an internal dynamic selector between Huff-0 and Range-0, developed by Y. Collet. Range-0 is based on the Range Coder algorithm from G. N. N. Martin's 1979 paper [21].
- **Zlib** [22] is an open-source data compression library written by Jean-loup Gailly and Mark Adler. When compiled with the #_Z_HUFFMAN_ONLY directive, the library runs in Entropy Coding mode.

Table II shows the decoding speed measured in MiB/s for two FCHD implementations, the 9-bit and 12-bit FCHD. From column 4 to 7, we display the decoding speed of the competitor software testes.

In the case of **Test data 1)**, which has the alphabet $A_\alpha = \{A, C, G, T\}$, depending on symbol frequencies, Huffman codeword bitlengths could range from 1 to 3 bits; This explains why the 9-bit FCHD is able to decode 3-9 symbols in a single decoding cycle, while the 12-bit version doubles that throughput (Fig. 2) and why the decoding speed of FCHD is always higher than other entropy coder implementations. The n-bit FCHD decodes a maximum of $n/2...n$ symbols in a decoding cycle, while Huff-0, Huff-X, Range-0 and others, zlib included, decode one single symbol per cycle.

Related to **Test data 4)**, we notice the 9-bit $FCHD$ is slightly faster than the 12-bit version. Here, the gaps in decoding speed are explained by the decoding table construction. Before decoding commences, FCHD must reconstruct the decoding table. The impact of reconstructing a larger table

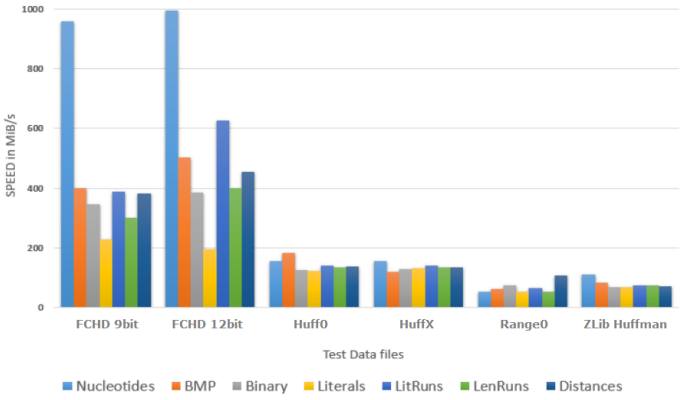

Fig. 3. Comparable decoding speed of FCHD versus other entropy coders, on all test data types.

vs a smaller table can be seen in Fig. 2. The 9-bit version is almost always faster than the 12-bit version.

Overall, our FCHD achieves up to 8.5 times faster decoding than its competition, while degraded compression ratios downfall 3% to 5% when compared to Huff-0 and zlib Huffman-only implementation.

Tests were performed using 64-bit software implementations running on a modern test computer equiped with Intel Xeon system CPU E5-1603, single threaded at 2.8GHz, 16GB RAM and high CPU priority-mode for applications. Decoding times represent time execution in milliseconds, counted as memory-to-memory (buffer to buffer) operations. Decoding speeds for Huff-0, Huff-X and Range-0 are timed internally and displayed while running.

V. CONCLUSIONS:

We found our Fast Canonic Huffman Decoder extremely suitable for limited-length Huffman codewords. Through empirical studies, we have learned that our FCHD method can perform Huffman decoding at speeds above 2GiB/s. We started with the classic Huffman tree from which we extracted the codewords' bitlengths. We have created a table with canonic codewords, constructed only from these bitlengths, following two simple canonical rules. Reparsing the input stream, we

have constructed a fast decoding table, by identifying segments of size N comprised of two elements, C_{vb} and R_{vb}. In the following step, the decoding table is stored in the output stream; this step is shown to have a minimum final impact due to its limited storage requirements. This is in line with what we have intended to accomplish with our newly devised Fast Canonical Huffman Decoding method. Empirically obtained test results showed 8.5 times faster speeds when compared to other competing entropy coders presented in this paper. Further minimizing the impact of larger decoding tables on compression ratio, when compared to lower-bit FCHD implementations, is one of our future research points.

REFERENCES

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Proc. of the IRE, Vol. 40, pp. 1098-1101, Sep. 1952.

[2] D. E. Knuth, "The Art of Computer Programming", Vol. 3, Sorting and Searching, Addison Wesley, Reading, MA, 1973.

[3] R.M. Fano. "The transmission of information", Technical Report 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., 1949.

[4] C. E. Shannon. "A mathematical theory of communication", Bell System Technical Journal, 27:379–423, 623–656, July, October 1948.

[5] Alistair Moffat, 2019, "Huffman Coding", ACM Comput. Surv. 52, 4, Article 85, August 2019

[6] Sieminskly, A. "Fast Decoding of the Huffman codes." Information Processing Letters 26 (1987/88) pp. 237-241 11 January 198

[7] Tanaka, H., "Data structure of Huffman codes and its application to efficient encoding and decoding, " IEEE Trans. Inf. Theory 33, 1 (Jan 1987), 154-156

[8] Renato Pajarola "Fast Prefix Code Processing", IEEE ITCC Conference, pages 206–211, 2003.

[9] Alistair Moffat and Andrew Turpin, "On the Implementation of Minimum Redundancy Prefix Codes", IEEE Transactions on Communications, Vol. 45, No. 10, October 1997

[10] Daniel S. Hirschberg and Debra A. Lelewer, "Efficient Decoding of Prefix Codes", Communications of the ACM, Vol.33, pp. 449-459, 1990

[11] Chung Wang, Yuan-Rung Yang, Chun-Liang Lee, Hung-Yi Chang "A memory-efficient Huffman decoding algorithm" Published in: 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1.

[12] Habib, A., Rahman, M.S. "Balancing decoding speed and memory usage for Huffman codes using quaternary tree". Appl Inform 4, 5 (2017).

[13] Swapna R. and Ramesh P., "Design and Implementation of Huffman Decoder for Text data Compression", International Journal of Current Engineering and Technology, Vol.5, No.3 (June 2015).

[14] Yann Collet, Huff-0, https://github.com/Cyan4973/FiniteStateEntropy Accessed on: May 20, 2022.

[15] Yann Collet, zHuff, https://fastcompression.blogspot.com/p/zhuff.html Accessed on: May 20, 2022.

[16] Yann Collet, Zstd, https://facebook.github.io/zstd/, Accessed on: May 20, 2022.

[17] National Library of Medicine, National Center for Biotechnology Information, Bethesda, MD, USA, Available online at: https://www.ncbi.nlm.nih.gov/genomes/VirusVariation/Database/nph-select.cgi, Accessed on May 20, 2022.

[18] Edward R. Fiala and Daniel H. Greene, "Data Compression with Finite Windows", Communications of the ACM, Vol. 32, Issue 4, April 1989, pp. 490-505.

[19] Yann Collet, Range-0, http://sd-1.archive-host.com/membres/up/182754578/Range0v07.zip Accessed on: May 20, 2022.

[20] Yann Collet, Huff-X, https://fastcompression.blogspot.com/p/huff0-range0-entropy-coders.html, Accessed on: May 20, 2022.

[21] G. N. N. Martin, "Range encoding: An algorithm for removing redundancy from a digitized message", Video & Data Recording Conference, Southampton, UK, July 24–27, 1979.

[22] Jean-loup Gailly, Mark Adler, "Zlib library", Available online at: https://www.zlib.net/, Accessed on: May 20, 2022.