Advanced Canonical Huffman Decoder

Author: **Sabin Belu**

Doctoral School ETTI, Univ. Politehnica of Bucharest, Romania

Supervisor/Mentor: **Prof. Dr. Daniela Coltuc**

Faculty of Electronics, Tc. and Information Technology, Univ. Politehnica of Bucharest, Romania

ACKNOWLEDGEMENT

The work has been funded by the Operational Programme Human Capital of the Ministry of European Funds through the Financial Agreement 51675/09.07.2019, SMIS code 125125

Advanced Canonical Huffman Decoder

ABSTRACT

We propose a novel method to decode canonical Huffman codes specifically designed to decode multiple symbols in a single decoding step. The encoding method used is the classic canonical Huffman encoding. Our decoding method is specifically designed with speed in mind and highly optimized for big throughput of decoding data, reaching 2.1GBytes/s on Intel i5 Core when decompressing high redundant data. At a maximum codeword length of 8 bits, our method requires a minimal amount of very memory for storing decoding tables and still operates at very high speeds. At our best knowledge, this is a trade-off we do not find in any of the published methods. We will also show that for Huffman codes that extend for up to 12 bits, the memory size and construction times of the decoding tables are negligible.

Index Terms— Canonical code, Huffman code, length-limited code, minimum redundancy codes, prefix codes, text data compression, text data decompression.

1. INTRODUCTION

Huffman codes have been around since 1951 when they were invented by D.A.Huffman. The problem he tried to solve was finding the most efficient method of representing any symbol, numbers or letters, from an input source, into a compact binary form. Huffman solved this problem with a simple idea, an idea that will place him into the highest Information Theory rank of underpinning technical achivers of all times.

Donald E. Knuth of Stanford University, the author of the multivolume series ***The Art of Computer Programming 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. D.A. Huffman's professor is also known to be the co-author of a less optimal method of encoding symbols, but still as famous huffman method. The method is called Shannon-Fano after its inventors, Claude Shannon and Robert Fano.

Shannon-Fano coding was developed in 1944 independently by Claude Shannon and Robert Fano . It emplies a greedy strategy by dividing the list of symbols to encode in two sublists with similar sum of symbol frequencies as close as possible. This strategy also emplies a top-down approach but does not usually produce a better and more optimal code than huffmans', which is a a bottom up procedure.

In theory, huffman codes approach optimality but when it comes to implemention, the method is much less optimal especially when fewer symbols are encoded. And this will be explained below:

Huffman method implies the following steps, when encoding:

1. Counting input symbol frequencies or assigning probabilites to each symbol.

2. Creating the huffman binary tree; each symbol will have a leaf node allocated.
3. Assigning codewords to symbols located in leaves, following a set-notation when traversing the binary tree: bit 0 for left traversal bit 1 for right traversal.
4. Creating the output stream by replacing the input symbols with their respective codewords by traversing the tree from the root to each symbol leaf.

In order to perform decoding, Huffman steps are:

1. The input symbol frequencies or assigned probabilites must be recovered from the transmission channel, or read from the stored source.
2. The huffman binary tree is constructed such as it matches the tree from the encoding step
3. The codewords are assigned to symbols located in leaves, following the set-notation decided in the encoding. bit 0 for left tree traversal and bit 1 for right tree traversal.
4. Creating the output stream by replacing the input symbols with their respective codewords

Huffman coding and in particular, the decoding part, needs to have the exact same information in order to reconstruct the original input stream; in order to achieve this, step 1 is the hardest to implement in a optimal fashio due to storing or transmission of such mandatory information over an information channel.

This is where the classical Huffman coding is far less optimal in practice then theory. The method proves to be less scalable for fewer symbols, due to the additional information required by step 1 and 2. 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 an optimal storage or transmission, or the omission of such. New progressive codes and coding method had to be found in order to optimally solve this problem and a new set of huffman codes that follow certain rules have been lately proposed.

Such codes are called the Canonic Huffman codes, and they possess unique properties that will allow us to solve step 1 and 2, the transmission or 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 sequencial. They will be discussed in more details during this article, but it is worth mentioning that the most important property of such codes has dramaticaly changes 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. For example, for a codeword of bits 3, the canonical codes will be 000, 001, 010 and 111 respectively. There is no longer necesarry to store the huffman tree in order to recontruct it before the decoding process, and pick the left subnode or the right subnode of the current node based on the input bit.

Given the auto generation of sequential codes, an obvious table can be created to dramatically increase the time of decoding such codes. Of course, this sounds simpler than it is because the codewords will eventually have a different number of bits. It’s where all these decoding methods come into place.

Therefore, we propose herein a brand new decoding method based on the canonical codes sequential property and also on the notion of intantaneous codes, that will be further described in details. We have designed this algorithm with speed in mind and with the intent to scrape all the extra steps taken to further ‘reconstruct’ a input symbol or portions of it completely guided by the KISS principle. Its optimized compiled code reaches very high speed during our testing and outperformed all the known huffman canonic implementations. Tests were conducted on the same test machine, under the same test conditions.

2. RELATED WORK

In [7], A. Sieminsky proposed 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 ‘0112’, called herein a group of bits, as the suffix for a codeword plus a codeword for another symbol. It could also 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, DC. The decoding context, for example, ‘101’, will determine the decoding table, DT, 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 .

$$DT\ Size \quad = \sum_{k=0}^{n} \mathrm{DC}\ x \sum_{k=0}^{n} \mathrm{DT}$$

Where

$$\sum_{k=0}^{n} \mathrm{DC} \quad is\ the\ number\ of\ all\ possible\ decoding\ contexts\ of\ arbitrary\ number\ of\ bits,$$

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.

In [8], 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 then needed to decode a single Huffman symbol.

In [2], Renato describes 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 precomputed 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 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 = 2 bits, since it also includes the ROOT as stated by Renato: "leaf nodes have all the same node transition tables as the root". For k >= 8 bits, the table storage and memory requirements could reach O(256 * 28 * 8) * sizeof(transition node), which we note to be prohibitive! Also, 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 main decoding steps. 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 algoritm could achive in decoding time.

Alistair Moffat and Andre Turpin propose in [6] 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 [6], 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 sequences called *strings*. In this initial phase, the algorithm compresses the input source by replacing sequences of symbols, herein named *strings*, with pointers to a dictionary. The formed dictionary is a collection of N *strings* of variable 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, their Huffman decoding algorithms presented in [6] do not work with symbols, but with a collection of “strings”, preparsed 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. Therefore, it still operates at a one symbol level. Memory Requirements: Method A1 requires 3*n*2 bytes and Method B1 2*n+54 in a typical application. Method B1 seems to have greater storage requirments 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 inital -phase dictionary which has to be recreated, with extra time and space requirements.

ACHD

Unlike any of the classical implementations that we have surveyed, our ACHD implements a proprietary algorithm that allows very fast decoding of *multiple symbols in one decoding cycle.*

Canonical Huffman is meant to ‘correct’ two major deficiencies of classic Huffman coding (Ref [1]).

1. Huffman Tree storing and transmission – this can be an ‘expensive’ step in terms of storage requirements and transmission of its content even for a small set of input symbols.

2. Traversing for decoding the Huffman tree

The classic tree traversal, 1 bit at a time, choosing either left of right child node depending on the bit value, is extremely exhausting in terms of decompression speed

As compared to the classic *method, the Canonical Huffman algorithm still relies on the Classical Huffman tree construction that is necessary to determine the codewords length. The implementation follows a set of rules relying only on these lengths.* Thus, in the decoding phase, the canonic codewords can be then regenerated following the same set of rules *and only the associated codewords length*, as compared with the clasical huffman, where the actual huffman tree needs to be present in the decoding step as well.

We begin our ACHD description with a canonic huffman description to show how these codewords are created.

CANONICAL HUFFMAN ALGORITHM DESCRIPTION

Suppose the following source alphabet s1, s2, …, sN, ordered after their frequency (s1 is the most frequent), and the corresponding codewords leghts l1<=l2<=…<=lN derived from classic Huffman tree generation.

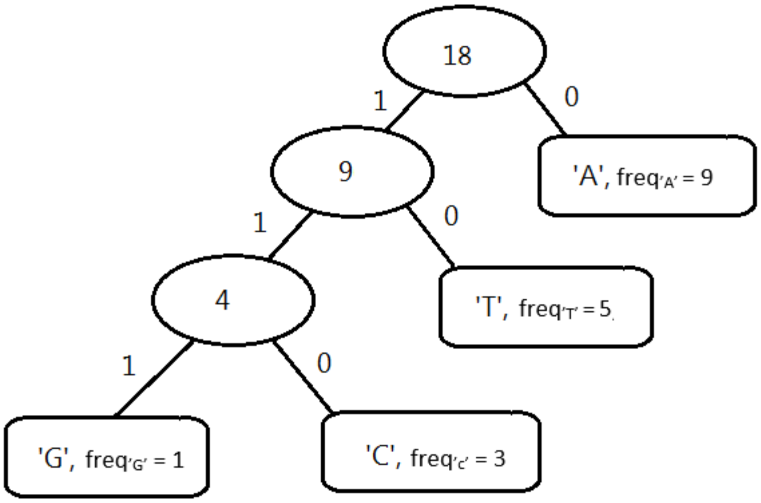
Canonical Huffman coding canon is described as following

Step 1. code = 0
Step 2: next_code = (code + 1) << (l2 – l1)

The algorithm always start with the initial code initialized with zero, at step 1. We iterate through all the input symbols and generate the next codeword by using the formula at step 2.

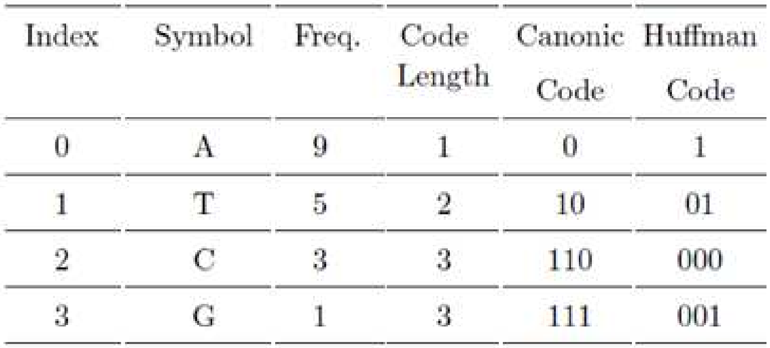
These rules will be used to regenerate the codewords in decoding phase, by simply using the codeword length. In our example, we assume the classic Huffman tree is constructed from the following input stream of symbols: "AAAAAAAAATTTTTCCCG" with the following symbol frequencies: freq$_{'A'}$ = 9 , freq$_{'T'}$ = 5, freq$_{'C'}$ = 3, freq$_{'G'}$ = 1. We proceed with the construction of the huffman tree by using the symbol frequencies alone. The first two lowest frequency symbols are picked, 'G' and 'C', to form a new node of frequency 4, both of their frequencies cummulated. The nodes 'G' and 'C' are removed from the list of symbol frequencies and the next two lowest frequencies are considered, the frequency of the node and the frequency of 'T'. The steps are repeated until the frequency list is empty.

Based on the input symbol frequencies, the following huffman tree is created:



Symbol ‘A’ has a codeword length of 1 bit, $Cw_{'A'} = 1$
Following the same notation:
$Cw_{'T'} = 2$
$Cw_{'C'} = 3$
$Cw_{'G'} = 3$
We note this list the *codeword lengthlist.*

The classic Huffman code is obtained by following the branch succession from the root to the symbol to encode. Classic Huffman usually codes succession to the left with bit 0 while succession to the ‘right’ with bit 1; following the algorithm steps, canonic Huffman codes left successions with 1 and right successions with 0. These notations do not affect the codeword lengths. They are the same in both constructions. Codeword values are different. This is explicitly shown below.
For the same example, the canonical Huffman code is built as follows:
Step 1 A=0 -> 0
Step 2 T=0+1=1 <<(2-1) -> 10
C=2+1=3 <<(3-2) -> 110
G=6+1=7 <<(3-3) -> 111
The codes are shown in **Table 1:**



Index	Symbol	Freq.	Code Length	Canonic Code	Huffman Code
0	A	9	1	0	1
1	T	5	2	10	01
2	C	3	3	110	000
3	G	1	3	111	001

Table 1: Symbol Frequencies and Huffman codewords

As classic Huffman, the canonic one is optimal since the codeword lengths are unchanged. It is also a prefix code due to the construction rule. Canonical Huffman codes have a series of properties that makes these codes very suitable for fast decoding

The most remarkable is the Consecutive-Values property.

For any given bit length l, l > 1, the associated codewords C(l) = {C1(l), C2(l) … CN(l)} represent a consecutive range of positive integers, meaning that

$$\forall i = 1..N, \mathrm{Ci}(\mathrm{l}) > 0$$

and

$$\mathbf{C1(l) < C2(l), C2(l) < \dots CN(l)}$$

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) as* the largest codeword of bitlength k+1,

$$\forall \boldsymbol{k},\ \boldsymbol{v} = \boldsymbol{C}(\boldsymbol{k}+\mathbf{1}),\ \mathrm{u} = \boldsymbol{C}(\boldsymbol{k}),$$
$$\text{then}\quad \mathrm{u} = \left(\frac{\boldsymbol{v}+\mathbf{1}}{\mathbf{2}}\right)$$

We propose a fast multi-symbol huffman canonic decoding solution implemented using a single lookup table. It requires minimum time for construction and it will be stored in the final output file. This lookup table will be further refered to as the ACHD decoding table. The formation of this table starts the moment the huffman output stream is constructed. Huffman output stream consists in codewords of variable length determined by the canonic Hufffman algorithm and ACHD decoding table that replace the input symbols. ACHD decoding table is stored in the archive, due to its minimalistic size and in order to reduce the time alloted by the decoder to reconstruct it

Construction of ACHD decoding table

In order to implement our ACHD algorithm, we partition the huffman binary output stream in segments that follow two rules:

A**. The greedy rule:** each segment tries to maximize the number of instantantly decodable codewords it contains with the condition that

B. The limitation rule: their cummulated codeword length is less or equal to an arbitrary chosen N.

For simplicity of examples, we have chosen to present this algorithm for N=8. Using the canonic huffman codes from table 1, we replace the symbols with their associated codewords and start identifying the segments.

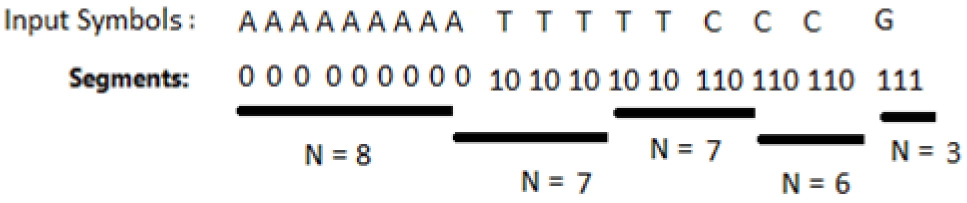
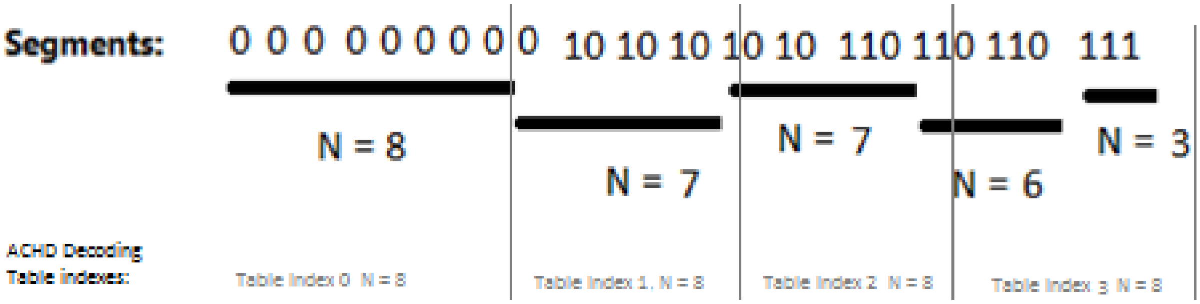


Fig. 3: Creation of decoding segment

Calculating the cummulated codeword length by counting the bits necessary for step B, we identify the following values, noted in binary: 0 0 0 0 0 0 0 0 2, 0 10 10 102, 10 10 1102, etc. Is it noticeable that the second segment identified by value 0 10 10 102 does not include the next codeword value 102 since it will contradict the second rule B, which states the cummulated codeword lengh has to be less than or equal to 8. We then use the segment value to generate the ACHD table index. This index is generated from a segment and a remainder of bits, if any. The remainder exists only if the cummulated codeword length is higher than N. Remainder has the following value:

$$\mathbf{Remainder_{bits} = C_{bv} - N}$$

ACHD table index is always of length N bits.



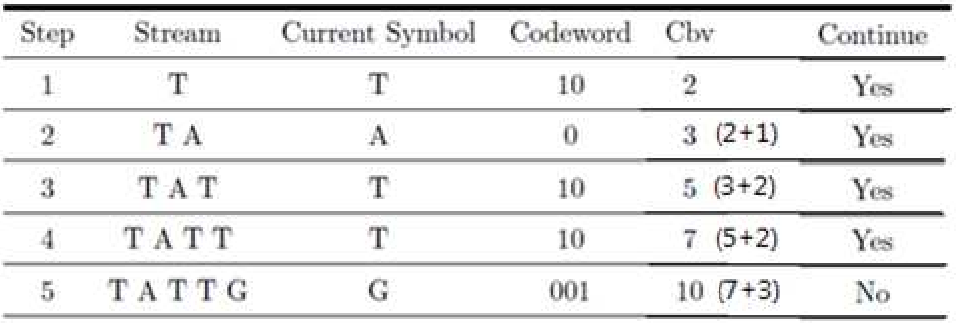
Using ACHD table index, we then store the aditional information required for fast decoding in a location of the decoding table where the ACHD table index points to.

This additional information needed for the decoding step is made of:

i. cummulated codeword length
ii. the input symbols associated with the instantly decodable codewords it contains
iii. the number of symbols from ii

Cummulated Codeword Length(i)	Symbols(ii)	Number of symbols(iii)	Segment Values
8	AAAAAAAA	8	00000000_2
7	ATTT	4	0101010_2
7	TTC	3	1010110_2
6	CC	2	110110_2
3	G	1	111_2

Table 4: Complete decoding table



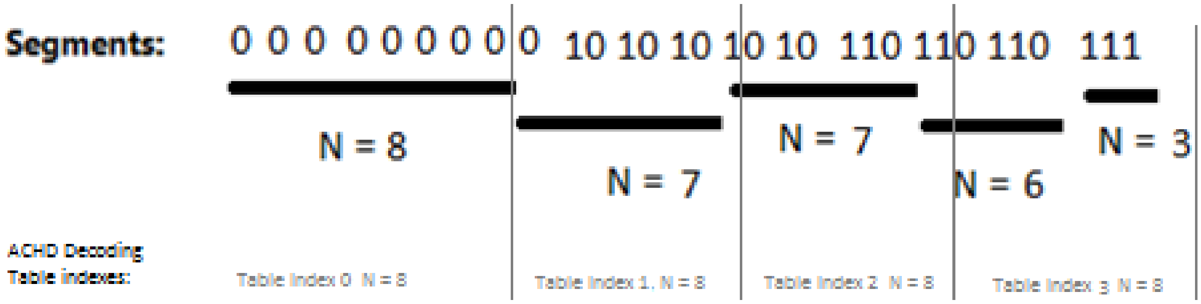
Step	Stream	Current Symbol	Codeword	Cbv	Continue
1	T	T	10	2	Yes
2	T A	A	0	3 (2+1)	Yes
3	T A T	T	10	5 (3+2)	Yes
4	T A T T	T	10	7 (5+2)	Yes
5	T A T T G	G	001	10 (7+3)	No

Table 5: Segments and Indexes

The remainder is a left over from the next instantly decodable codeword which is not fully used because of rule A

For instance, the second segment 0 10 10 10 2, has the remainder bit 12. The segment value and the remainder bit, they form the ACHD table index 0 10 10 10 12. As states above, ACHD table index must always be of length N.

The ACHD table index is formed out of a segment and a remainder of bits, if the current codeword length is less than N . ACHD table index is always of length N bits.



Huffman output stream, together with the additional information needed to successfully decode it forms what we call 'the archived file' or simply 'archive''. In order to minimize the size of the archive that may be stored or transmitted over a channel, the additional information must be kept at a minimum.

Using canonic huffman coding, only bit lengths of symbols (table 1, codelength column) need to be stored or transmitted in order to allow successful reconstruction of codewords. This has a huge potential of optimizations as compared with the classic method, where the entire huffman tree had to be stored or sent through a channel in order to perform decoding. It is one of the reasons why canonic huffman coding is the prefered method in the industry over the huffman classic method;

In order for perform successful decoding, the archive must contain:

- the bit length for all symbols as defined in Table 1 under Code Length
- ACHD decoding table (see table 4: i, ii, iii)
- Huffman output stream

ACHD Experimental results

Our Advanced Canonical Huffman Decoder has been tested against different data types such as nucleotides, bitmap files, text and binary files. Other test files relate to an in-house LZFG data compression program we have developed, such as literals (uncompressed symbols), literal runs, lengths, length runs and distances (substring copies coded as back reference pairs of distance and length tokens). All together, they form some specific test files that we use in our tests.

Short descriptions of LZFG-A markers and tokens:

- **Literals:** Literals are uncompressed symbols.
- **LitRuns:** LitRuns are used to decode 1 or more literals. A LitRun indicates that 1 or more literals are available in the stream at the current step.
- **LenRuns:** LenRuns are used to decode 1 or more length of a substring copy (values and count)
- **LitRuns and LenRuns are coded on 4-bits and together they form a 1-byte marker. The byte is then decoded in two 4-bit values. A LitRun and a LenRun.**
  - If a Run (LitRun or LenRun) value is 0x0F it is considered a marker and two more bytes are read; this keeps the Runs to a maximum value of 65535 + 0x0F.
  - If a Run value is less than 0x0F then it is a token and more bytes are read from the input stream. The number of bytes read is exactly the ‘Run’ value.
- **Substring copies:** LenRuns are used to decode 1 or more substring copy length. A substring copy, also called match, is a string that has been previously seen in the input stream and it is coded as <distance, length> pair.

We have used test data sets which we believe to be generically applicable especially for an order-0 entropy coder, as explained in details:

Data File Types	Software TEST File	Test Data Size (bytes)	Bytes per symbol	Description
1	Nucleotides	106,459,204	1	A,T,C,G symbols in random order (patterns do not count in order-0 encoding)
2	BMP	131,036,917	3	24-bit bitmap file
3	Binary	587,209,776	1	Precompiled header files from Visual Studio compilations and other windows x86 (32bit) binary files
4	Literals	172,040,192	1	uncompressed symbols (symbols not included in a match)
5	LitRuns	78,864,384	1	tokens and markers related to literal runs
6	LenRuns	78,864,384	1	tokens and markers related to length runs.
7	Distances	315,445,248	4	copy substring positions encoded as back reference (distance) from the current input pointer position

8	8-symbol random file	10,485,760	**1**	8-symbol random file artificially generated
9	16-symbol random file	10,485,760	**1**	16-symbol random file artificially generated
10	32-symbol random file	10,485,760	**1**	32-symbol random file artificially generated
11	64-symbol random file	10,485,760	**1**	64-symbol random file artificially generated
12	128-symbol random file	10,485,760	**1**	128-symbol random file artificially generated

Table 6 – Data File Types used for testing

Test data 1) is made of highly compressible data, only four symbols thus a shorter alphabet with random symbol frequency distribution. Test data 2) is made of highly compressible data but contatins more than 1 symbol with only 1 symbol being more frequent. The most frequent symbol comes from coding the white color from the bitmap background. Test data 3) is mostly made of of binary data, with a uniform distribution of frequencies among all 8-bit ASCII symbols. It is a mixture of text source file paths, source file contents, indexed content, inversed indexes, hashes, temporary compilation binary data, compiled data, etc. Test data files 4) through 7) are artificial files, created by the LZFG–A1 data compression algorithm. The LZFG-A1 method is known to output <literals> and <length, distance> pairs. They are coded with the help of other tokens and markers called LitRuns an LenRuns.

Depending on their values, LitRuns and LenRuns are split between tokens and markers, each of them being locally adaptive. Encoding them further and independently result in better compression ratio then having their probabilities mixed up into a single stream of data.

Test Comparisons with other encoders

We have compared our ACHD implementation against the following publicly available entropy coders:

- Huff0 – developed by a FACEBOOK developer named Yann Collet. An order-0 Huffman entropy coder that is using canonic Huffman codes. This application is currently used in large scale applications like ZStd and ZHuff
- Range0 - Order-0 range coder developed by Yann Collet
- HuffX – a mixed entropy coder, a fusion of huff0 and range0, with a dynamic selector, developed by Yann Collet.
- Zlib - written by Jean-loup Gailly (compression) and Mark Adler (decompression) running in canonic huffman mode only (_Z_HUFFMAN_ONLY parameter present)

The reasons for chosing the above software applications to perform tests against our ACHD is that the internet is not abundant in good, high-speed implementations of canonic huffman – only algorithm. One of the first implementation of this algorithm was noticed in the zlib/gzip application libraries. The entropy coding portion was written by Mark Adler, an American software developer more famous for his Adler-32 checksum algorithm.

We've also chosen Zlib because of its ubiquitousness these days. There are few applications that we know including data compression and not using among other methods, the zlib compression. **Zlib** is so widespread that even PkWARE decided to implement in their own deflate algorithms the streaming capabilities of Zlib, modifications that were eventually written by the author of this article. **Zlib** is a combination of dictionary coding and entropy encoding methods, a mixture of LZ77 and canonic huffman. Since comparisons with a dictionary coder wouldn't have been relevant, we've used during testing an in house 64 bit built of Zlib and supplied the engine with the _Z_HUFFMAN_ONLY parameter. This ensures the use of the order-0 entropy coding engine, only, thus making it suitable to be inserted into our comparison tests.

Huff0, HuffX and **Rang0** were downloaded from their authors' website titled RealTime Data Compression. They were the candidates for the "real time" software library that Yann Collet was developing for facebook, called ZStandard. They seem to embody the state of the art in entropy coding from the world wide web.

Huff0 is an order-0 block-based entropy coder which employs canonic huffman algorithm. The input data is divided 16KB blocks, while the compressed output consists in a tree followed by compressed entry. The author states that "smaller blocks allow faster adaptation, but cost more in header, so a trade-off had to be chosen." Apparently, the 16KB size has been selected as the optimal block size for distribution of input statistics.

Range0 was added as a comparison between different entropy methods, as it employs the range coding algorithm. It was discovered by Nigel N. Martin in 1979 and employs a FIFO arithmetic coding method that was first introduced by R. Pasco, in 1976. In Range0, the input data is divided into 128KB chunks. The compressed output consists into frequency count followed by compressed data. The presence of the frequency count data implies the software uses static adaption modeling. The author states he has chosen 128KB of data for better adaption and better compression results, in most cases, in comparison with the 16KB of Huff0.

HuffX, is in short an implementation of a mixed entropy coder between huff0 and range0, with a dynamic selector; an entropy adaptation as it called by the author of this application. HuffX seems to compute the "cost" of entropy encoding by analyzing eight successive chunks of 16KB of data, encoded with the Huff0 algorithm, and the cost of a single 128KB block encoded by Range0. The method with the lowest transmission cost per block is chosen and the software advanced into the input stream by analyzing the next 128KB of data.

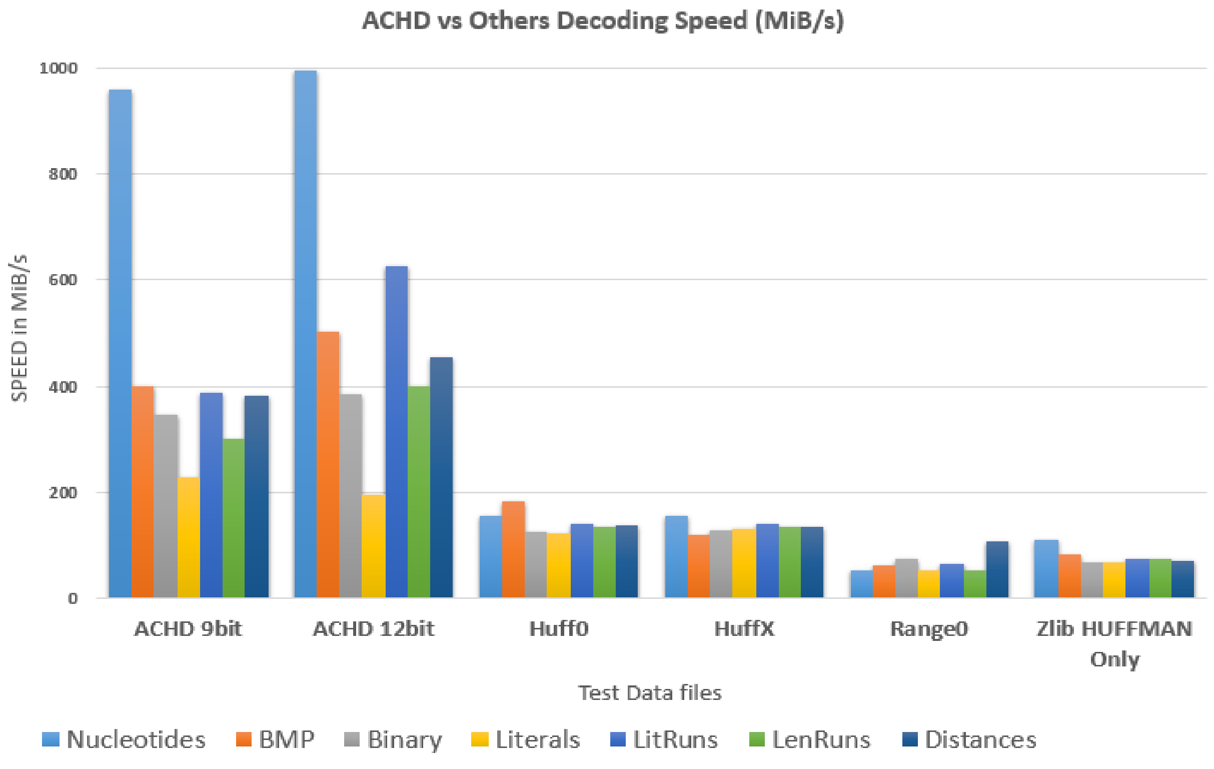
Here are the complete test results:

		Decoding Speed(MiB/s)					
Test #	**TestFile**	**ACHD 9bit**	**ACHD 12bit**	**Huff-0**	**Huff-X**	**Range-0**	**Zlib HUFFMAN**
1	Nucleotides	966.93	985.71	162.51	169.55	60.46	110.24
2	BMP	400.53	501.81	182.64	114.36	59.24	83.42
3	Binary	344.83	385.41	127.21	128.93	76.44	69.16
4	Literals	230.11	196.11	124.66	131.37	54.34	69.61
5	LitRuns	389.69	626.71	140.51	142.42	67.42	75.29
6	LenRuns	299.65	370.51	129.72	133.91	54.11	74.11

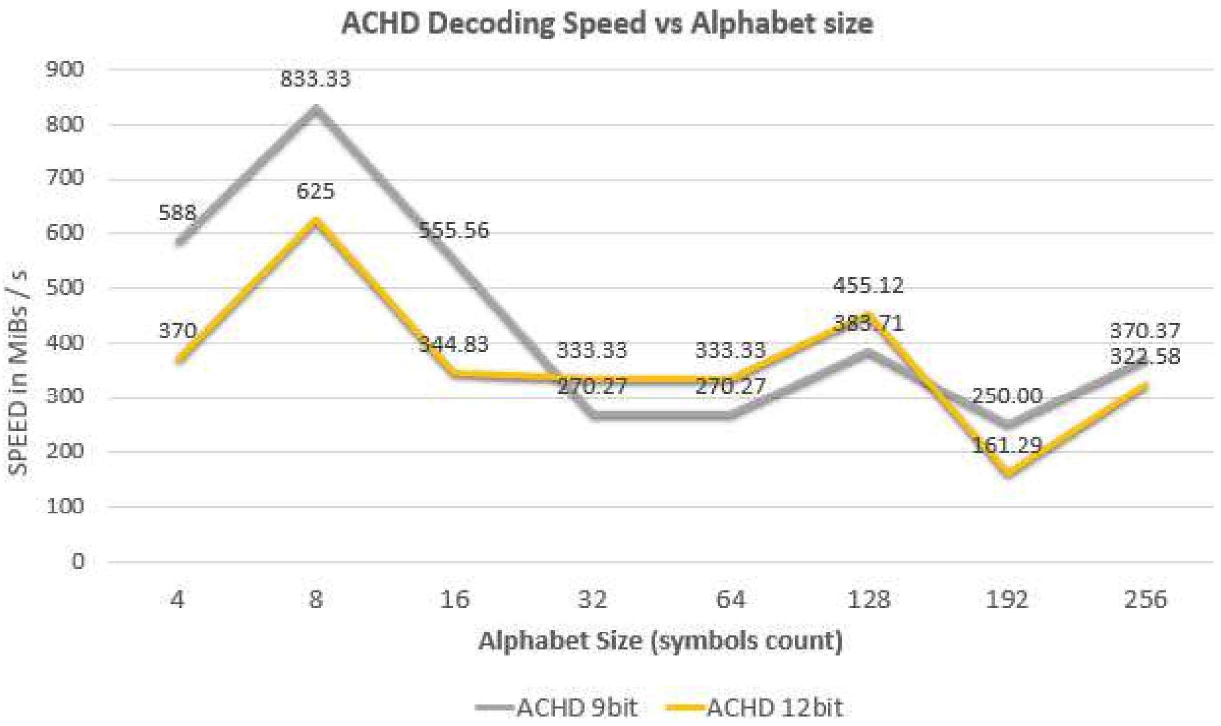
7	Distances	383.71	455.1	137.6	134.8	108.11	70.62
8	8-symbol random file	833.33	625.01	172.00	173.10	60.60	90.91
9	16-symbol random file	555.56	500.83	173.10	173.10	61.60	126.58
10	32-symbol random file	270.27	333.33	170.60	169.30	58.90	106.38
11	64-symbol random file	270.27	322.58	165.50	165.50	58.90	70.62
12	128-symbol random file	383.71	169.51	137.60	134.80	108.10	70.62

Table 7 – Complete test Results, Decoding Speed in MiB/s and milliseconds

Graphics:



Graphic 1 – The decoding speed charts of ACHD and others entropy coders on all test data types



Graphic 2 – How the alphabet size affects the decoding speed of ACHD and others entropy coders

Interpretation of results:

We have conducted a series of experiments on various data types and source files. The main purpose of these tests is to show the improvement in decoding time performance of the presented ACHD canonical huffman decoding algorithm and how it compares with other entropy codecs performance. For reference, we have also included comparisons with the Range-0 application written by Yann Collet, available on his site [12], which is entirely based on the range coding algorithm, a new entropy coding method devised by G. Nigel N. Martin [17].

For the first data test case, a four letter alphabet is to be encoded; a maximum of 2 bits of information per symbol is needed to represent a four letter alphabet symbol; the 9 bit ACHD is therefor able to output 4 to 9 bytes in a single decoding cycle, while the 12 bit ACHD may be able to output 6 to 12 bytes in a single decoding cycle.

The speed result here is simply explained by the architecture of the decoding algorithm itself. A 12 bit ACHD decodes a maximum of 12 symbols in a decoding cycle, while Huff0, HuffX, Range0 and Zlib they all decode a single symbol per cycle.

Test case #4, the 9bit ACHD is slightly faster than the 12bit version. This result is explained by the uniform distribution of statistics among these uncompressed symbols. They tend to have only order-0 correlation. The difference in speed is also given by the decode table construction, which is 1.314ms in the first case as compared to 12.881ms in the second case.

In all the above case, our ACHD achieves up to 8.5 times faster decoding than its counterparts while degraded compression ratios are limited to 3% compared to Huff0 and 5% compared to zlib's 'huffman only' implementation.

Testing has been done under an Intel Xeon system CPU E5-1603 with 16MB RAM at 2.8GHz, single threaded, high CPU mode for applications. The software has been compiled for 64bit. Decoding or decompression speed represents the speed counted as memory- to-memory (buffer to buffer) decoding. Huff_0 decoding speed is timed by the program itself. Zlib timing counted internally by the main compiled application using ZLib.

Conclusions:

Our new and Advanced Canonic Huffman Decoder, ACHD, is extremely suitable for canonic huffman codes with an average codeword bit length of up to 12 bit.

Through empirical studies, we have learned that our ACHD method is able to decode symbols at speeds at about 2GB/s and above while processing highly redundant files.

ACKNOWLEDGEMENT

The work has been funded by the Operational Programme Human Capital of the Ministry of European Funds through the Financial Agreement 51675/09.07.2019, SMIS code 125125

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. Julia Abrahams, “HUFFMAN CODE TREES AND VARIANTS,” DIMACS,Rutgers University, Piscataway,NJ.
3. R. G. Gallager, Information Theory and Reliable Communication, Wiley, New York,
1968.
4. R. Sedgewick, “Algorithms”, Addison-Wesley, Reading, MA, 1983.
5. D. E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching,
Addison Wesley, Reading, MA, 1973.
6. Renato Pajarola “Fast Prefix Code Processing”, IEEE ITCC Conference, pages 206–211, 2003.
7. 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.
8. Habib, A., Rahman, M.S. “Balancing decoding speed and memory usage for Huffman codes using quaternary tree”. Appl Inform 4, 5 (2017).
9. 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).
10. Alistair Moffat and Andrew Turpin, “On the Implementation of Minimum Redundancy Prefix Codes”, IEEE Transactions on COMMUNICATIONS, Vol. 45, No. 10, October 1997
11. Daniel S. Hirschberg and Debra A. Lelewer, “Efficient Decoding of Prefix Codes”, Communications of the ACM, Vol.33, pp. 449-459, 1990
12. Sieminskly, A. “Fast Decoding of the Huffman codes.” Information Processing Letters 26 (1987/88) pp. 237-241 11 January 1988
13. 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
10. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. “*Managing Gigabytes*”. New York: Van Nostrand Reinhold, 1994. ISBN 9780442018634. pp. 33-35.
11. Jean-loup Gailly, Mark Adler, “Zlib library”, available on: https://www.zlib.net/
12. Yann Collet, Range-0, January 3, 2011, https://fastcompression.blogspot.com/2011/01/range0-new-version-v07.html
13. Yann Collet, Huff-0, January 3, 2011, https://fastcompression.blogspot.com/2011/01/quick-loock-back-at-huff0-entropy-coder.html
14. Yann Collet, Huff-X, March 26, 2011, https://fastcompression.blogspot.com/2011/03/best-of-both-worlds-mixed-entropy.html
15. J. van Leeuwen, “On the construction of Huffman Trees”, Automata Languages and Programming, Third Intl.Colloquium at the Univ. of Edinburgh, edited by S. Michaelscon and R. Milner, 20.21.22. 23. July 1976, http://www.staff.science.uu.nl/~leeuw112/huffman.pdf
16. Richard Clark Pasco , "Source coding algorithms for fast data compression", Stanford, CA 1976
17. G. Nigel N. Martin, Range encoding: An algorithm for removing redundancy from a digitized message, Video & Data Recording Conference, Southampton, UK, July 24–27, 1979.
18. Variable length binary encodings. Gilbert E. N. , and Morre E. F., Bell system tech. Journal, vol 38, 1959, pp 933-967.
19. Generalised Kraft Inequality and Arithmetic Coding, Rissanen J. J. , IBM journal of R&D, May 1976, pp 198-203