An Algorithm for Combinatorial Entropy Coding

Stephan Bärwolf

Integrated Communication Systems Group
Ilmenau University of Technology
Ilmenau, Germany
stephan.baerwolf@tu-ilmenau.de

***Abstract*—Entropy coding (esp. order-0) was one of the first techniques for lossless data compression, dating back to the invention of modern information theory. Over such a long period of time different schemes were invented and entropy coding has experienced various improvements: Huffman published its minimal tree structured codes and then Witten, Neal and Cleary presented a scheme leading to even better results.**
While entropy compression is still used today in most of recent compression schemes, it has not lost its significance. This paper presents an encoding and its corresponding decoding algorithm not using trees or intervals to do entropy compression. Instead it derives permutations from the input which are mapped to natural numbers. Furthermore this paper gives an impression about the compression performance by comparing some first results with well known entropy compression schemes.

Keywords- entropy; coding; data compression

I. INTRODUCTION

Today, nearly all lossless and even lossy compression schemes are using at least a build-in order-0 entropy coder. Because normally entropy coding is very easy to understand and very effective in compressing non uniform distributed inputs, it is a preferred "last-stage" compression technique in such schemes. Since Shannon posted his first ideas about compression, known as Shannon-Fano, in his famous paper [3], different concepts for entropy compression have emerged: Some years after Shannon, David Huffman [4] improved Shannons scheme. Still using the same concept of binary trees, Huffman changed the way of constructing the tree structure and proved it to be optimal. Finally, in 1987 a paper [5] was published, which clearified an algorithm leading to nearly always better compression results than Huffman. This breakthrough was done by using successive bisection of an interval instead of trees to generate code words.

As already mentioned, even if today's compression schemes use more advanced algorithms, one of the previous mentioned entropy coders (often Huffman) is still part of them. For example, the Microsoft LZX [7] extends the idea of LZ77 [6] by utilizing entropy compression for match-lengths and -positions via Huffman codes.
An extreme example of entropy compression used today are the Burrows-Wheeler transform (BWT) [8], its bijective version BWTS [9], and other sort transforming modifications [11]. Since the BWTs are only "transformations", the whole compression effect is done (after some intermediate processing stages) in one final entropy compression stage [10].

This paper presents a different concept for order-0 entropy compression by mappings of permutations to enumerations and vice versa. Instead of using trees or intervals, the compression results of the presented technique therefore should never be worse than the one compared to arithmetical coding.

The paper is structured as following. A simple algorithm for encoding is presented and discussed in the first section. After this section the same is done for the decoding. In section 4, some first results are presented by using the usual compression corpora ([14],[15],[16]). Finally, the paper will be closed with a conclusion/future work chapter.

II. ENCODING

The idea of encoding an input word "I" over the finite, non-empty alphabet "A" ($I \in A^*, (a_0, a_1, \ldots, a_{d-1}) \in A, a_0 < a_1 < \cdots < a_{d-1}$), is to enumerate its represented permutation under its given symbol frequencies α ($[\alpha[a_0], \alpha[a_1], \ldots, \alpha[a_{d-1}]] = \alpha \in (\mathbb{N} \cup \{0\})^k, \alpha[a_i] = |I|_{a_i}, n = |I| = \sum_{k=0}^{d-1} \alpha[a_k]$).
Furthermore a_k will be synonymous with k.

Because such an enumeration would be bounded by a multinomial coefficient (1), the enumeration could be stored with only $log_2(\binom{n}{\alpha[\ ]})$ instead of $n \cdot log_2(d)$ bits.

$$\frac{(\sum_{k=0}^{d-1} \alpha[k])!}{\prod_{k=0}^{d-1} \alpha[k]!} = \frac{n!}{\prod_{k=0}^{d-1} \alpha[k]!} = \binom{n}{\alpha[\ ]} < d^n \qquad (1)$$

For example, the word "mississippi" leads to enumeration 32592 (out of $\frac{11!}{4!\cdot 1!\cdot 2!\cdot 4!} = 34650$), where $\alpha["i"] = 4, \alpha["m"] = 1, \alpha["p"] = 2, \alpha["s"] = 4$, see table I.
A different word may produce the same result, if its permutation is the same and just its symbol frequencies α are different: The word "MISSISSIPPI" leads to same enumeration 32592.

 ISBN: 978-1-61208-322-3

TABLE I. ENUMERATIONS FOR A GIVEN α

enumeration	word
0	iiiimppssss
1	iiimippssss
...	
99	mppiisisiss
100	imppsiisiss
...	
1999	iippisisssm
...	
32591	imssissippi
32592	mississippi
32593	msisissippi
...	
34649	sssppmiiii

Algorithm 1 shows a way to efficiently calculate such an enumeration.

First for every symbol $a = a_i$ (see line 5) a separate, partial enumeration ("$code_a$") is generated by just taking symbols larger or equal to a_i into account.
Because permutations of smaller symbols $a_j, a_j < a_i$ have already been processed, they are ignored in further iterations. Therefore, $code_a$ is a sum of binomial coefficients $\binom{n}{m}$ for every position where "a" occurs in "I". Therefore, "n" is the number of symbols (till the current processed position) greater or equal to "a" and "m", the count of already processed occurrences of "a".
This "outer" loop is done backwards in order to enable forward decoding.

The final output result, "$code$", is the combined value of all "$code_a$".

$$base_a = \prod_{i=0}^{a-1} \binom{n - \sum_{l=0}^{i-1} \alpha[l]}{\alpha[i]}, base_0 = 1 \quad (2)$$

$$\begin{aligned} code &= code_0 + \sum_{a=1}^{d-1} code_a \cdot \prod_{i=0}^{a-1} \binom{n - \sum_{l=0}^{i-1} \alpha[l]}{\alpha[i]} \\ &= \sum_{a=0}^{d-1} code_a \cdot base_a \end{aligned} \quad (3)$$

III. DECODING

In order to decode a given enumeration, first the original symbol frequencies α must be known to the decoder. In a practical application, this is either already known (due to special constructions), or has to be transmitted to the decoder separately.
Within the further text it is assumed to know the correct α before decode.

If α is known, then $n = |I|$ can be retrieved efficiently, because $n = \sum_{k=0}^{d-1} \alpha[k], d = |\alpha|$.
Knowing α also enables the decoder to calculate each $base_a$ using (2) and therefore each $code_a$ using (3). Resolving the equation (3) for $code_a$ leads to (4).

$$code_a = \left\lfloor \frac{code \bmod (\prod_{i=0}^{a} base_i)}{\prod_{i=0}^{a-1} base_i} \right\rfloor \quad (4)$$

Algorithm 1 Derive an enumeration (code) from an (input) word

Require: $msg \Leftarrow$ to be encoded message ($msg = I$)
Require: $\alpha \Leftarrow$ frequency of each character (byte) in the message
Require: $n \Leftarrow$ size of decoded message ($n = \sum_{a=0}^{255} \alpha[a]$)
1: *// initialize output:*
2: $code \Leftarrow 0$
3: $bytes_{done} \Leftarrow 0$
4: *// process position of every value "a" individually*
5: **for** $a = 255$ **down to** 0 **do**
6: $\quad bytes_{done} \Leftarrow bytes_{done} + \alpha[a]$
7: $\quad bytes_{relevant} \Leftarrow 0$
8: $\quad bytes_{processed} \Leftarrow 0$
9: $\quad code_a \Leftarrow 0$
10: $\quad code \Leftarrow code * \binom{bytes_{done}}{\alpha[a]}$
11: $\quad$ *// loop over message, track positions of value "a", ignore smaller values*
12: $\quad$ **for** $msg_{position} = 0$ **to** $(n-1)$ **do**
13: $\quad\quad$ **if** $msg[msg_{position}] \geqq a$ **then**
14: $\quad\quad\quad$ **if** $msg[msg_{position}] = a$ **then**
15: $\quad\quad\quad\quad bytes_{processed} \Leftarrow bytes_{processed} + 1$
16: $\quad\quad\quad\quad code_a \Leftarrow code_a + \binom{bytes_{relevant}}{bytes_{processed}}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad\quad bytes_{relevant} \Leftarrow bytes_{relevant} + 1$
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: $\quad code \Leftarrow code + code_a$
22: **end for**
23: **return** $code$

To retrieve a position for symbol "a" from $code_a$, the biggest position possible (with a binomial coefficient still fitting into $code_a$) has to be successively substracted, as indicated in line 17 of algorithm 2. Thanks to $\alpha[a]$ the decoder already knows the right count of positions to decode.
Because such decoded positions were positions in the set of unprocessed symbols (symbols greater or equal) they have to be transformed into a global array index (lines 19 to 26).

Finally, all indices of the "msg" array have been processed and msg contains the decoded "I".

 ISBN: 978-1-61208-322-3

Algorithm 2 Retrieve word from its enumeration

Require: $code \Leftarrow$ to be decoded number
Require: $\alpha \Leftarrow$ frequency of each character (byte) in decoded message
Require: $n \Leftarrow$ size of decoded message ($n = \sum_{a=0}^{255} \alpha[a]$)
1: *// initialize output:*
2: $msg \Leftarrow$ each byte filled with value 255, $|msg| = n$
3: $bytesleft \Leftarrow n$
4: *// process permutation of every char individually:*
5: **for** $a = 0$ **to** 255 **do**
6: *// retrieve permutation code for positions of value a*
7: $code_a \Leftarrow code \bmod \binom{bytesleft}{\alpha[a]}$
8: *// update code for succeeding permutations*
9: $code \Leftarrow code \ div \ \binom{bytesleft}{\alpha[a]}$
10: $position \Leftarrow bytesleft$
11: *// to track the unprocessed indices within msg:*
12: $msg_{position} \Leftarrow n$
13: $msg_{unprocessed} \Leftarrow bytesleft$
14: *// start decoding positions where value "a" occurs in message*
15: **for** $k = \alpha[a] - 1$ **down to** 0 **do**
16: $maxpos \Leftarrow position$
17: $position \Leftarrow$ find biggest m with $k \leqq m < maxpos$ and $\binom{m}{k+1} \leqq code_a$
18: $code_a \Leftarrow code_a - \binom{m}{k+1}$
19: *// place value "a" into msg at unprocessd position "position"*
20: **while** $msg_{unprocessed} > maxpos$ **do**
21: $msg_{position} \Leftarrow msg_{position} - 1$
22: **if** $msg[msg_{position}] = 255$ **then**
23: $msg_{unprocessed} \Leftarrow msg_{unprocessed} - 1$
24: **end if**
25: **end while**
26: $msg[msg_{position}] \Leftarrow a$
27: **end for**
28: $bytesleft \Leftarrow bytesleft - \alpha[a]$
29: **end for**
30: **return** msg

IV. First results

The results in table II (resp. table III) show the compressed size and the percentage to the original size of the Huffman-, arithmetical- and the presented scheme.
In order to use a set of representative files, the Canterbury [16] (resp. Calgary [15]) corpus was used.
There were no other preprocessings than the direct entropy compression with the mentioned schemes.
In all schemes it was assumed that decoders will have full apriori information about α and used this apriori information at the beginning of encoding.

For generating Huffman compressed files, the open source "libhuffman" [17] was used. This encoder operates in two phases, the first one scans the input in order to construct an optimal tree. The second phase uses this tree to compress the input byte by byte. For "libhuffman" the function storing the extra information about α (from phase one) was commented out in order to preserve comparability.

For generating arithmetical compressed files, the algorithm from Witten et al. [5] was used. It was slightly adapted to avoid using END-symbols and to work in two phases like "libhuffman" does. As in "libhuffman", the arithmetical encoding used the knowledge about α from the beginning, but also did not store any information about α to the output.

The algorithm of the presented scheme always stored $log_2(\binom{original\_size}{\alpha[\ ]})$ bit (ceiled up to the next full byte) as output. Again no information about α was put to the output.

From both tables it can be seen, that the presented scheme always is the best compressing one.

V. Conclusion

This paper presented a different concept for order-0 entropy compression, where mappings from permutations to enumerations are used. The paper presented also an algorithm for encoding and for decoding. First compression results were compared to two well established encoders, indicating promising compression performance since the presented scheme always is the most performant one.
Since it can be shown, that at most $(n+d) \cdot log_2(n+d) - n \cdot log_2(n) - d \cdot log_2(d)$ additional bits ($n = |I|, d = |\alpha|$) are required to be transmitted to the decoder for decoding the correct α, the presented coding scheme seems to have no issues affecting decodability.

The presented technique also offers promising possibilities for future work on this field:
Since binomial coefficients can be precalculated into some kind of cache or table, the scheme nearly hasn't any slow multiplication. Because the last remaining multiplication in line 10 of algorithm 1 could be replaced by much faster logical left-shifting - no multiplications or even divisions are necessary at all.
Furthermore and because of commutative multiplication/addition, the algorithms outer- and inner loops are good candidates for parallelization with nearly no synchronization needs and therefore nearly full speedup.

Acknowledgments

The author would like to thank the "Deutsche Telekom Stiftung" Germany [18] for its funding and for its ongoing efforts, actions and activities to support education and thus research and science. Furthermore the author also would like to thank the team of the Integrated Communication Systems Group at TU-Ilmenau for its general support at all times.

 ISBN: 978-1-61208-322-3

TABLE II. THE CANTERBURY CORPUS [16] COMPRESSION PERFORMANCE

filename	org. size (bytes)	huffman size	huffman %	arithmetical size	arithmetical %	presented size	presented %
alice29.txt	152089	87688	57.66	86837	57.10	86788	57.06
asyoulik.txt	125179	75806	60.56	75235	60.10	75187	60.06
cp.html	24603	16199	65.84	16082	65.37	16035	65.17
fields.c	11150	7026	63.01	6980	62.60	6936	62.21
grammar.lsp	3721	2170	58.32	2155	57.91	2126	57.14
kennedy.xls	1029744	462532	44.92	459971	44.67	459779	44.65
lcet10.txt	426754	250565	58.71	249071	58.36	249008	58.35
plrabn12.txt	481861	275585	57.19	272936	56.64	272880	56.63
ptt5	513216	106551	20.76	77636	15.13	77563	15.11
sum	38240	25645	67.06	25473	66.61	25353	66.30
xargs.1	4227	2602	61.56	2589	61.25	2559	60.54

TABLE III. THE CALGARY CORPUS [15] COMPRESSION PERFORMANCE

filename	org. size (bytes)	huffman size	huffman %	arithmetical size	arithmetical %	presented size	presented %
README	2479	1492	60.19	1483	59.82	1457	58.77
bib	111261	72761	65.40	72330	65.01	72273	64.96
book1	768771	438374	57.02	435043	56.59	434981	56.58
book2	610856	368300	60.29	365952	59.91	365877	59.90
geo	102400	72556	70.86	72274	70.58	72117	70.43
news	377109	246394	65.34	244633	64.87	244555	64.85
obj1	21504	16051	74.64	15989	74.35	15868	73.79
obj2	246814	194096	78.64	193144	78.25	192971	78.18
paper1	53161	33337	62.71	33113	62.29	33058	62.18
paper2	82199	47615	57.93	47280	57.52	47228	57.46
paper3	46526	27275	58.62	27132	58.32	27084	58.21
paper4	13286	7860	59.16	7806	58.75	7768	58.47
paper5	11954	7431	62.16	7376	61.70	7334	61.35
paper6	38105	24023	63.04	23861	62.62	23808	62.48
pic	513216	106551	20.76	77636	15.13	77563	15.11
progc	39611	25914	65.42	25743	64.99	25687	64.85
progl	71646	42982	59.99	42720	59.63	42668	59.55
progp	49379	30214	61.19	30052	60.86	30000	60.75
trans	93695	65218	69.61	64800	69.16	64734	69.09

REFERENCES

[1] D. Salomon, “Data Compression The Complete Reference,” 4th ed. London: Springer, 2007.

[2] D. J. C. MacKay, “Information Theory, Inference, and Learning Algorithms,” version 6.0 Cambridge University Press, June 2003, pp. 32.

[3] C. E. Shannon, “A Matematical Theory of Communication,” Reprinted with corrections from The Bell System Technical Journal, vol. 27, October 1948, pp. 379–423, 623–656.

[4] D. A. Huffman, “A method for construction of minimum-redundancy codes,” Proceedings of the I.R.E., September 1958, pp. 1098–1101.

[5] I. Witten, R. Neal, and J. Cleary, “Arithmetic Coding for Data Compression,” Communications of the ACM, vol. 30, no. 6, June 1987, pp. 520–540.

[6] A. Lempel and J. Ziv, “A Universal Algorithm for Sequential Data Compression,” IEEE transactions on information theory, May 1977, pp. 337–343.

[7] Microsoft, “Microsoft LZX Data Compression Format,” version 4.71.410.0 Microsoft Cabinet SDK, , March 1997.

[8] M. Burrows and D. J. Wheeler, “A block-soring Lossless Data Compression Algorithm,” System Research Center, Palo Alto, USA, research report, May 1994.

[9] J. Gil and D. A. Scott, “A Bijective String Sorting Transform,” Israel/USA, July 2009.

[10] J. Abel, “Improvements to the Burrows-Wheeler Compression Algorithm: After BWT Stages,” preprint Düsburg-Essen, Germany, March 2003.

[11] M. Kufleitner, “On Bijective Variants of the Burrows-Wheeler Transform.” Prague, Czech Republic: Proceedings of PSC 2009, pp. 65–79.

[12] A. Salomaa, “Public-Key Cryptography,” 2nd ed. Finland: Springer, 1990.

[13] M. Lothaire, “Combinatorics on words,” Reading, Massachusetts: Addison-Wesley, 1983.

[14] R. Arnolds and T. Bell, “A corpus for the evaluation of lossless compression algorithms.” Christchurch, NZ: University of Canterbury.

[15] T. C. Bell and I. Witten, “Calgary compression corpus,” [retrieved: December, 2013]. [Online]. Available: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/

[16] M. Powell and T. Bell, “The Canterbury corpus,” [retrieved: December, 2013]. [Online]. Available: http://corpus.canterbury.ac.nz/

[17] D. R. Richardson, “libhuffman - An Open Source Huffman Coding Library in C,” [retrieved: December, 2013]. [Online]. Available: http://huffman.sourceforge.net/

[18] http://www.telekom-stiftung.de

 ISBN: 978-1-61208-322-3