A Method for Virtual Extension of LZW Compression Dictionary

István Finta*, Lóránt Farkas*, Sándor Szénási† and Szabolcs Sergyán†
*Technology and Innovation, Nokia Networks, Köztelek utca 6, Budapest, Hungary
Email: istvan.finta@nokia.com
†John von Neumann Faculty of Informatics, Óbuda University, Budapest, Hungary
Email: szenasi.sandor@nik.uni-obuda.hu

***Abstract*—Lossless data compression is an important topic from both data transmission and storage point of view. A well chosen data compression technique can largely reduce the required throughput or storage need. As a tradeoff, data compression always requires some computing resources which are in a correlation with the achieved compression rate. For a particular use case the best suitable compression method depends on the statistical characteristics of the data, the applied computing paradigm and the data access pattern. In this contribution we will introduce an extension of LZW based compression technique which overall performs well in case when a relatively long substring occurs quite frequently in the file to be encoded.**

***Keywords*—Data Compression, Lossless Data Compression, Encoding, Decoding, Lempel-Ziv, LZW, LZ Family, LZAP, LZMW, Dictionary-based Compression**

I. Introduction

In telecommunication networks value added applications like OSS or CEM frequently require data transmission from network elements towards the (usually centralized) Operating Support System(OSS) or Customer Experience Management(CEM) application. Typically, such data takes the form of CSV messages where values are a-priori mapped to keys via configuration files. In other typical cases such data takes the form of html files, where the context of data is described in the data model synchronized 'out of band', before the actual data comes in. For instance the ID of a performance counter is specified so that the OSS or CEM application can find its meaning after an appropriate mapping of the ID to the definition of the counter.
Passing clean text CSV or html is inefficient. Considering the case of performance counters there may be thousands of them defined per network element type, therefore their transmission especially in the redundant html format with all the tags included in the content takes away considerable bandwidth from the usually expensive backhaul links. This is especially true in the next generation mobile networks where the number of base stations might be in the order of tens or hundreds of thousands.
In many cases mobile operators mandate network equipment vendors to store OSS or CEM data for several years in a queryable form. While aggregations make it possible to decrease the granularity of the data over time, in certain cases a drill down in historical data is required to reveal certain anomalies. The granularity of aggregated data is usually not sufficient for drill down operations. Therefore it is worth to consider the usage of lossless compression technique to increase the storage capacity for raw data.
Data compression is a widely used method to reduce the original size of the data to be stored or transmitted. There are two main types of compressions: lossy and lossless compression. In this paper an extension of a lossless compression algorithm will be introduced. From now on when we speak about compression we always mean lossless compression type, except when marked differently.
Compression method basics are covered by the discipline of information theory. Lossless compression methods can effectively compress low entropy files, where the entropy of the file is defined generally in the information theoretical sense as a measure of the amount of information that is missing before reception of the file.
According to [1] from algorithmic point of view there are three main compression approaches:

- Sliding Window Algorithms,
- Dictionary Algorithms and
- Non-dictionary Algorithms.

During our investigations we have examined the application of compression methods in big data environment for telco data and as a side effect we have discovered the there may be room for further development in case of LZAP and LZMW algorithms. However LZAP and LZMW are based on LZW. There are several LZW derivatives, where the principle is the same, but due to some modification in special cases the derivatives can achieve better performance. Lempel and Ziv invented the LZ77 and LZ78 dictionary type compression algorithms in 1977 and 1978. In 1984 Terry Welch modified the LZ78 algorithm and invented the LZW algorithm [2].
Therefore in the Background section LZW and related methods will be described in more detailed, where we will identify the development possibilities of the algorithm.

II. Background

A. LZW encoding

During encoding LZW maintains a dictionary in which the entries are divided into two parts. The size and content of the first part, which is mostly called initial part, is immutable and contains all the individual symbols from a pre-defined alphabet with a sequence number associated with the position. The second part is a dynamic one and contains at least two symbols long words over the alphabet. The numbering of the dynamic part begins from the end of the initial part without

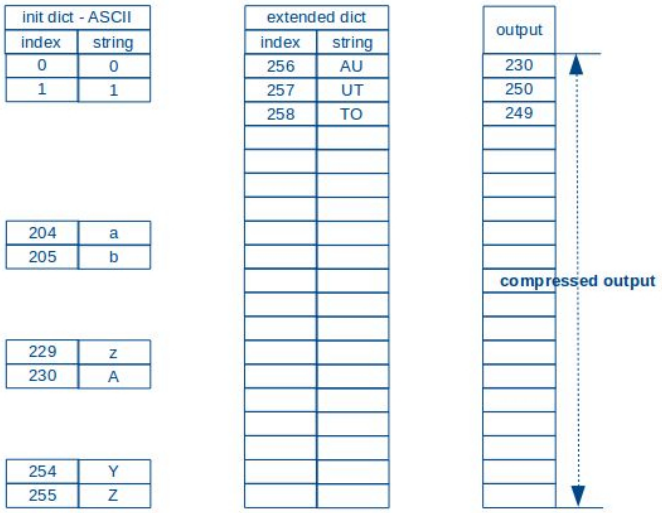

Fig. 1. LZW Encoding Example

overlapping.
Supposing that our alphabet is the set of ASCII characters and we have an input text to be compressed, the dynamic part of the dictionary is built up according to the following rule:

- The encoder builds words (-W_b) from the input text character by character and looks up W_b in the dictionary.
- The encoder builds W_b until it is not available in the dictionary, or when the encoder reaches the end of the input text. When the W_b is not in the dictionary this means that W_b is one symbol longer than the previous longest character sequence with the same prefix W_{cm}. W_{cm} is also called *current match*.
- W_b will be written into the first empty position of the dynamic part of the dictionary. Alongside the encoder issues the sequence number of W_{cm}.
- Then the encoder forms a new W_{nb} from the last character of W_b.
- Then swaps W_b with W_{nb}, drops W_{nb} and starts a new cycle.

When a dictionary gets full the dynamic part will be flushed out and rebuilt periodically to stay adaptive.
In Fig. 1 encoding of "AUTO..." input is visible.

B. LZW decoding

In case of decoding the decoder has to have the same initial dictionary. The decoder reads the issued sequence numbers. Based on the numbers and the static part of the dictionary the decoder is able to rebuild the dynamic entries. This information is enough to reconstruct the input text.

C. LZW Problem Scenarios

As it is visible from section II-A the dictionary is built quite slowly. This means that the encoder can increase the stored entries by one character compared to the previously longest prefixes. In case when a relatively long substring occurs quite frequently, due to the dictionary construction strategy, the full coverage of that particular substring may require at least as many entries as long the substring itself is(Problem_1/P1).
The situation is even worse if two frequently occurring substrings(W_1, W_2) differ from each other only in the first character. In this case, due to the dictionary construction, full coverage of W_1 and W_2 may require twice as much entries in the dictionary as if W_1 and W_2 were identical (Problem_2/P2).
Besides the above two scenarios supposing that the encoder is in the middle of the encoding of an input text and there is a recurring substring W_1, the encoder will find that particular substring in its dictionary (and therefore compress the input text efficiently) only if it can start the word parsing from exactly the same character as did it in previous case. It means that an offset between the previous and actual substring parsing may significantly decrease the quality of the compression (Problem_3/P3).

D. LZMW and LZAP encoding and problems

Let us define *previous match* as the preceeding entry in the dictionary relative to current match.
LZMW(MW:Miller, Wegman) [3] tries to increase the hit ratio by inserting into the dictionary the concatenation of the previous match and current match. The main problem with this method is that it consumes the entries faster than LZW. Other problem is that encoding side time complexity is high compared to LZW.
LZAP(AP:All Prefixes) [4] is a derivative of LZMW and tries to resolve P1, P2 and P3 according to the following: during dictionary building besides the full concatenation of previous match and current match the extended previous matches are also stored. Extensions here mean all prefixes of the current match. That is why one match will occupy as many entries in the dictionary as many symbols reside in the current match. This approach can significantly increase the hit ratio, however it is too greedy from memory consumption point of view.

III. Methodology

The goal is to eliminate somehow the memory consumption problem of LZMW or LZAP. To solve this problem a new approach will be introduced which we will call Virtual Dictionary Extension(-VDE). VDE from processing point of view resides between LZMW and LZAP. With Virtual Dictionary Extension we will be able to increase the hit ratio compared to LZW, but this method will require only as many entries as LZW.

To make it possible in the dictionary we have to distinguish the positions of the entries from their indexes/sequence numbers. In case of LZW, LZMW or LZAP the position of an entry is identical with its index. In those cases the distance between two adjacent entries is one. In the followings dictionary entries will be called *primary entries* and will be denoted by p. The idea is that in case of VDE the distance between two adjacent primary entries is one in terms of position but can be greater in terms of indexes. The position associated indexes will be denoted by i_p. The indexes which fall between two i_p will be denoted by i_v(virtual index). Virtual indexes, without position in the dictionary, refer to

composite or virtual entries. That is why dictionary extension is called virtual. During encoding the indexes will be emitted instead of positions(as happened in case of LZW, LZMW or LZAP). The applied composition rule must consider that at decoding side we have to be able to reproduce the original input from the mixture of position associated and virtual indexes. Apart from this boundary condition we can choose any composition rule which fits to our problem domain. In the followings we will show the Linear Growth Distance(-LGD) composition rule.

A. Linear Growth Distance composition rule

As previously mentioned the dictionary has an initial part and a dynamic part. Supposing that we have an alphabet which resides in the initial part of the dictionary. The initial part is immutable therefore in the followings we can consider it as a constant offset from both position and index point of view. To make the introduction of VDE-LGD encoding easier we ignore the initial part caused offset and focus only on the usage of dynamic part.
In case of LGD we can count the position associated indexes according to the following formula:

$$i_p = \frac{p * (p+1)}{2},$$

which is nothing else but the triangular number [5]. Considering the linearly growing number of i_v between i_p, which is always equal with the number of preceeding primary entries, with i_v we can refer to concatenations which are generated from words of previous primary positions. With this technique we can increase the hit ratio with identical number of entries.

Let's see an example: the text to be compressed is let's say: "*asdfasdr*". Based on the composition rule the following words will be available:

0 - as , a
1 - sd , s
2 - asd
3 - df , d
4 - sdf
5 - asdf
6 - fa , a
7 - dfa
8 - sdfa
9 - asdfa
10 - asdr , asd
11 - fasdr
12 - dfasdr
13 - sdfasdr
14 - asdfasdr

The primary entries are marked with bold. The emitted symbol itself is displayed after the comma instead of the index of the emitted symbol.

B. Linear Growth Distance Encoding

To explain encoding let us first compare the content of LZW(left column) and VDE-LGD(right column) dictionaries and the emitted indexes based on the previous example:

0 - as , a	0 - *as* , a $\rightarrow i_p$
1 - sd , s	1 - s*d* , s $\rightarrow i_p$
2 - df , d	3 - df , d $\rightarrow i_p$
3 - fa , f	6 - fa , f $\rightarrow i_p$
4 - asd , as	**10 - asdr , asd** $\rightarrow i_v(=2)$

To determine the indexes lets consider the bold "asdr" row. In the legacy case "as" would be the current match. We propose to start examine after the "as"(marked by italic) match the successive primary entry without the first character, which is in this case "sd" without "s", that is "d"(marked by italic). In case of matching one takes the next primary entry, "df", and performs the previously mentioned examination again, "f"(marked by underline) in this case. However the next symbol in the input text to be encoded is "r", so the extension process stops here. When the last match has been reached it counts the Number of Hops(NoH) and maintains the first match. The index to be sent out will be computed according to the following rule:

- if the first match is the last match, so there is no subsequent match, the index is an i_p type and counted based on the dictionary position,
- if the first match differs from the last match the index to be sent is computed according to this:

$$i_v = i_p + (p_l - p_f),$$

where

- p_l is the position of last match, and
- p_f is the position of first match.

The original LZW algorithm requires the following modifications:

- First we have to introduce a new, so called, buffer area to be able to simulate and handle the subsequent word comparison failures. This solution makes it possible to continue the process in the middle of the "next entry", in case of comparison failure, without information loss.
- The second difference is that we have to distinguish from searching point of view the first match from subsequent matching(s).
- The third difference is that it has to differentiate the initial part of the dictionary from the dynamic part. In case of LDG virtual extension will be applied exclusively to the dynamic part of the dictionary.

C. Linear Growth Distance Decoding

At decoding side the reconstruction of the input works like the following: when an index arrives - denoted by i_a - the algorithm examines if it is a primary entry or not. To perform this the following formula is used:

$$p_c = \frac{-1 + \sqrt{1 + 8i_a}}{2}.$$

From here there are two main scenarios possible:

- In case when the p_c is an integer without remaining value this means that the dictionary entry searched for is a primary entry. It is possible to look up the entry from the dictionary directly.
- Otherwise take the floor function of the computed position, signed p_f. This will provide last primary entry of match. Then compute the base index from the position, signed with i_b, with the following formula:

$$i_b = \frac{p_f * (p_f + 1)}{2}.$$

Then with a simple subtraction it is easy to define the NoH = $i_a - i_b$. With this information step back NoH and start to generate the derivative entry. From here, if the word is computed, the process continues as in case of the original LZW algorithm.

There is only a small difference compared to the original decoder method when the referenced primary entry still not present: it only can takes place when it depends on the previous primary entry. To compute the missing reference entry simply step back with NoH, which is practically 1 in this case. Then take the first character of that primary entry as an addition to the previously used entry, no matter if it is a derivative or primary one. Then this combined entry will be the missing referenced entry that have to be written into the dictionary. From here every step takes place according to has been written before.

IV. Results

In the following we will present some experimental result regarding LGD. Here we have to note that this is a proof of concept implementation and lacks lots of optimization possibilities, but can provide a good overview.

A. Test Environment and Benchmarking

In actual implementation the ASCII-8 has been selected as initial part of the dictionary.
In practical implementations indexes are represented with dynamic length. This means that in order to increase the compression gain, based on the actual position within the dictionary the number of bytes used to store the dictionary entry index are aligned. It must be highlighted that during our testing NO dynamic length codeword optimization was used. Therefore the comparison of this solution to the original method is referenced to not-optimized implementations but it can give good impression about the compression gain. It is important that both implementations flush out the dictionary. However, due to LGD, we had to set the fix codeword length as high as to cover maximum of computed index length. If we used the same fix length for legacy encoder it would give false result because there is no need to use as long codewords as one required due to LGD. To resolve this contradiction the legacy implementation will appear with two codeword length representations:

- with an optimal one, and
- with the same one used by LDG implementation.

B. Data characteristics

Two sources/inputs have been selected to test the compression performance:

- a 310 KBytes CSV file containing customer experience-related log data (e.g. unsuccessful call), and
- a 333 KBytes html file representing a performance measurement log from a network element.

C. Results of VDE-LGD

First let's see the CSV related compressions at TABLE I.

TABLE I. Result of CSV compression

	Original Size - M [Kbytes]	Number of Positions	Codeword [bytes]	Compressed Size - C [Kbytes]	C/M
legacy LZW	310	4096	3	148	0,477
legacy LZW	310	4096	2	99	0,319
LGD	310	4096	3	79	0,254

As it is visible LGD performs better all over. Even if legacy LZW used 2 bytes long index representation according to the second row.

TABLE II. shows the HTML file compression result:

TABLE II. Result of HTML compression

	Original Size - M [Kbytes]	Number of Positions	Codeword [bytes]	Compressed Size - C [Kbytes]	C/M
legacy LZW	333	4096	3	228	0,684
legacy LZW	333	4096	2	152	0,456
LGD	333	4096	3	117	0,351

The characteristic of the result is similar: VDE-LGD performs better than the legacy LZW method.

Now let's see the speed related measurements. In this case a ten executions based average is counted and shown in the TABLE III. and TABLE IV. with both CSV and HTML inputs.

TABLE III. Result of CSV compression duration

	Encoding [ms]	Decoding [ms]	Sum [ms]	LZW Based Relative Duration
legacy LZW	1336.6	1020	2356.6	1
LGD	1318.2	1106.1	2424.3	1.028

TABLE IV. Result of HTML compression duration

	Encoding [ms]	Decoding [ms]	Sum [ms]	LZW Based Relative Duration
legacy LZW	2022	1368	3390	1
LGD	1484.8	1431.9	2916.7	0,860

V. Conclusion

As it is visible from the results the compression gain is not for free. At encoding side more computation, comparison may required than in case of traditional LZW. But the applied internal buffer can probably balance the increased number of comparisons, since LGD encoding always performs better than LZW encoding. At decoding side the computation overhead, which is caused by the calculations and the concatenations, is constantly present and clearly visible from the result tables.
However both side requires further investigations because as we noted before these are Proof of Concept implementations. The applied data structure can highly influence the performance related results. Nevertheless the tests are proved that with LGD we can achieve better compression ratio with relatively small performance loss on decoding side. Besides that the overall performance of LGD can better in case of write_once_read_once environment, like stream processing frameworks(Storm [6] for instance). The overall performance can also better in write_once_read_many_times environment(like Hadoop Distributed File System [7]) considering the achieved compression gain and if the number of read operations are limited below a threshold.

In *LZW Problem Scenarios* subsection we have used terms like ”quite frequent” and ”relatively long”. The positive experiments inspire us further investigation regarding these terms. We plan to examine if it possible to introduce objective measures for this terms and investigate the relation to existing ones like entropy or information content.

We also plan to examine other distance functions between position associated indexes and their processing complexity on both encoding and decoding side, especially in write_once_read_once and write_once_read_many_times environments.
However, it is visible if we would use constant distance(-CD) with value two instead of LGD, then we could give back LZMW.
This means that construction of VDE is backward compatible with some existing compression method since it forms an abstraction layer.

Other to consider is the optimal dictionary or buffer size: this may depend from both the statistical attributes of the input and the applied data structures.

References

[1] *History of Lossless Data Compression Algorithms*, http://ethw.org/History_of_Lossless_Data_Compression_Algorithms, last visited 2015-09-08

[2] Terry Welch: *A Technique for High-Performance Data Compression*, IEEE Computer Society Journal Volume 17 Issue 6, pp 8 - 19, June 1984

[3] David Salomon, Giovanni Motta: *Handbook of Data Compression*, 5th edition London, England: Springer-Verlag, 2010, pp. 377-378.

[4] David Salomon, Giovanni Motta: *Handbook of Data Compression*, 5th edition London, England: Springer-Verlag, 2010, pp. 378-379.

[5] *Triangular Number*, http://mathworld.wolfram.com/TriangularNumber.html, last visited 2015-10-15

[6] *STORM - A distributed realtime computation system*, http://storm.apache.org/documentation/Home.html, last visited 2015-10-15

[7] *HDFS Architecture*, https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, last visited 2015-10-15