DNA Inspired Bi-directional Lempel-Ziv-like Compression Algorithms

Attiya Mahmood, Nazia Islam, Dawit Nigatu, and Werner Henkel
Jacobs University Bremen
Electrical Engineering and Computer Science
Bremen, Germany
Emails: {a.mahmood, n.islam, d.nigatu, w.henkel}@jacobs-university.de

***Abstract*—The bi-directional reading processes in DNA replication and gene expression together with the similarities between the so-called alternative splicing and Lempel-Ziv (LZ) algorithms has motivated us to incorporate bi-directional readings into LZ algorithms. LZ77, LZ78, and LZW84 are universal lossless data compression algorithms. A modified version of these algorithms that takes into account both forward and reverse readings is presented in this work. It is shown that bi-directional reading can improve the compression ratio at the expense of slight modifications in LZ algorithms provided that there exists some symmetry in the information content. Results are presented for text, image, and audio files.**

I. Introduction

LZ77 [1] and LZ78 [2] are lossless data compression algorithms proposed by A. Lempel and J. Ziv in 1977 and 1978, respectively. LZW84 [3] is a similar algorithm proposed by T. A. Welch in 1984. LZ77 is based on a sliding window approach, where compression is achieved by preserving the retransmission of a data vector that has already been sent. It maintains a search buffer which stores the previously sent string of characters and when any new upcoming string matches the symbols already stored in the search buffer, the position and length of that match are sent. At the decompression end, a similar receive buffer is maintained which retrieves data from the position specified by the compression end. The overall idea of LZ78 is the same but it maintains an adaptive dictionary to store the previously processed symbols and compression is achieved by sending the index of the dictionary, where a match occurs. LZW84 is similar to LZ78, however LZW84 has an initialized dictionary and the output does not contain a newly occurring character to update the decompressing dictionary. In all of these algorithms, input data statistics and compression are done in a single pass.

We have modified the original LZ77, LZ78, and LZW84 algorithms for bi-directional reading and analyzed their compression performance. The idea for bi-directional reading emanated from studies of the DNA, which is sometimes read by enzymes in both forward and reverse direction on opposite strands leading to different proteins. Additionally, in vertebrates, a so-called alternative splicing of the RNA is in place, which in a way looks related to LZ77. This made us consider source coding algorithms for bi-directional reading. We have investigated LZ77 and LZ78, LZW84 algorithms for variable window lengths and variable numbers of bits to address the dictionaries, respectively. The LZ like algorithms might be further modified to suit DNA compression by introducing base pair conjugacy relations as additional matching criterion.

In the following sections, we first describe the bi-directional and Lemple-Ziv like processes in the DNA and then we introduce the bi-directional algorithms along with their corresponding pseudo-codes. In the third section we demonstrate and discuss simulation results. Finally, conclusions are presented.

II. Bi-Directional reading and Lempel-Ziv like processes in the DNA

DNA is a double-stranded molecule. Each strand has so-called $3'$ and $5'$ ends. The two strands are anti-parallel with the so-called leading strand oriented in $3'$ to $5'$ direction, whereas the lagging strand runs from the $5'$ end to the $3'$ end [4]. This labeling gives DNA/RNA a built-in directionality. In between the two ends, the building blocks called nucleotides exist. The nucleotides are made of a sugar phosphate backbone and one of the four nitrogenous bases attached to the sugars. These bases are called Adenine (A), Thymine (T), Cytosine (C), and Guanine (G). Adenine is always paired with Thymine and Cytosine is always paired with Guanine.

During DNA replication, after the two strands are separated to serve as a template for producing a copy, an enzyme called DNA polymerase reads the strands in the $3'$ to $5'$ direction placing the corresponding nucleotides along the way (see Fig. 1). On the leading strand template, addition of complementary nucleotides is continuous. However, in the lagging strand, the DNA is read in opposite direction (for DNA polymerase, reading only makes sense in the $3'$ to $5'$ direction) and hence, DNA synthesis occurs in short and separated fragments (Okazaki fragments)[5].

The other biological process involving the bi-directional reading is gene expression. Gene expression is a process by which the genetic information from a segment of DNA, called genes, is used to synthesize protein or in some cases a functional RNA [4]. The making of a protein consists of two steps, transcription and translation, in eukaryotes, there is an intermediate step entitled RNA processing (splicing). During transcription, the gene sequence is copied into Messenger RNA (mRNA) using the template strand of the DNA. The enzyme involved in this reaction is known as RNA polymerase. Here, also RNA polymerase reads the sequence in the $3'$ to

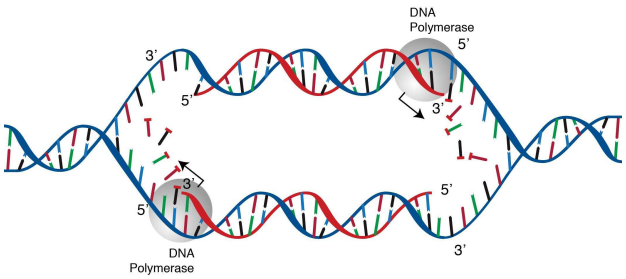

Fig. 1. DNA replication (http://www.genome.gov/dmd/img.cfm?node=Photos/Graphics&id=85151)

5′ direction. Hence, depending on the location of the gene, RNA polymerase reads and copies the content in forward or backward direction. The information in the mRNA sequence is read in triplets, called codons, by a large and complex molecule called ribosome. The codons are then translated to an amino acid sequence based on what is known as the genetic code. The codon ATG signals the start of the translation process in the 5′ to 3′ direction and the process stops when one of the three stop codons is recognized (TAA, TAG, and TGA). Finally, the amino acids are connected by a polypeptide chain and folded into a protein.

Genes can also exist in overlapping positions of the opposite DNA strands. In such cases, reading in the opposite directions leads to a synthesis of different proteins. For instance, in E. coli, the gene "htgA" is embedded within "yaaW" as shown in Fig. 2. Reading in forward direction, starting from ATG (shown in blue arrow) the ribosome continues reading until a stop codon occurs, in this case TGA. Using the same sequence, reading in opposite direction on the other strand (red arrows), the ribosome will find the start codon CAT (complementary to ATG) and reads and decodes the corresponding protein until a stop codon TAC (complementary to TGA) is encountered.

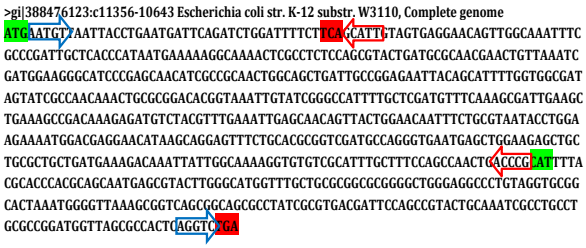

Fig. 2. Genes "htgA" and "yaaW" in overlapping position on opposite strands (nucleotide sequence of gene "yaaW" of E. coli K12.)

In eukaryotes, the RNA copied from the DNA has to be further processed. The premature mRNA contains exons which code for protein parts and introns, which are not considered. Hence, before the RNA is translated, the introns are removed and the exons are spliced together [6]. In a regulated process called alternative splicing, some of the exons may either be incorporated in the final matured mRNA or will be left out just as the introns. This process gives the possibility of creating multiple, but related, proteins from relatively smaller DNA segments. The steps are illustrated in Fig. 3. If one thinks of the introns to be already existing dictionary entries and the exons to be currently not needed dictionary components, directing which nucleotide sequences to be added for the to be produced protein, the described alternative splicing technique somehow resembles Lempel-Ziv (LZ77) source coding [1].

Together with overlapping (bi-directional) genes on both strands, the question arises, how to make LZ algorithms to work in a bi-directional fashion, too.

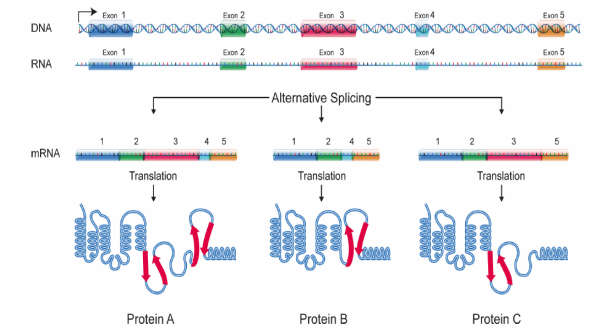

Fig. 3. Alternative splicing (http://www.genome.gov/Images/EdKit/bio2j_large.gif)

III. Bi-Directional Algorithms

A. Modified LZ77 Compression Algorithm

In the conventional LZ77 algorithm, the entire data are traversed in the window of search and lookahead buffers. Compression is achieved by sending the position and length inside the search buffer, where the characters match exactly the currently processed string of characters present in the lookahead buffer. Our bi-directional reading mechanism modifies the matching criteria of this search buffer. The original pseudo-code of LZ77 [7] is modified to include bi-directional reading as follows: The longest match of a string is found in both forward and reverse direction. If no match exists, then only the first non-matching character (F_S) is sent, otherwise the position (Pos), the length (L) of the maximum matching string in either direction, a flag bit (F), and the next non-matching character (N) in the lookahead buffer (LA) are output by the compressor. The flag bit specifies the direction of the match to be either forward or reverse. Thus, the numbers of coded

Algorithm 1 : Bi-directional LZ77

while LA(Lookahead Buffer) $\neq$ Empty **do**
  $L_1 \leftarrow$ Length of Forward match;
  $L_2 \leftarrow$ Length of Reverse match;
  $L \leftarrow \max(L_1, L_2)$;
  if $(L == 0)$ **then**
    Output $\leftarrow(0, 0, F_S$ in LA);
    Shift left the window by 1 character;
  else
    Output $\leftarrow$(Pos, F, L, N in LA);
    Shift the window left by$(L + 1)$ positions;
  end if
end while

bits (L_{C}) needed to compress the data in this modified scheme are:

$$L_{\mathrm{C}} = \log_2(n - L_{\mathrm{A}}) + \log_2(L_{\mathrm{A}}) + 1 + N_{\mathrm{S}}, \qquad (1)$$

where n is the total window length containing both search and lookahead buffers, L_{A} is the length of lookahead buffer, the flag bit needs one bit, and N_{S} is the number of bits needed to represent the next symbol. With respect to the original algorithm, the only additional cost is a flag bit, which we will see not to have a detrimental effect on the performance.

B. Modified LZ78 Compression Algorithm

In the original LZ78 algorithm, the dictionary is gradually built at both the compression and the decompression end. At every instant, input data symbols are searched at already stored dictionary indices. If a subset of data symbols exists in dictionary entries, then the reference of that dictionary index is sent followed by the next non-existing symbol. The set of matching symbols and the next non-existing symbol are also added at the next available dictionary index. The original pseudo-code of LZ78 [7] is modified to include bi-directional reading as follows:

Algorithm 2 : Bi-directional LZ78

```
W1 = NIL (for forward matching);
W2 = NIL (for reverse matching);
while input available do
    K ← next symbol from input;
    if W1K exists in dictionary then
        W1 ← W1K;
    end if
    if W2K exists in flipped(dictionary) then
        W2 ← W2K;
    end if
    if (W1K does not exist in dictionary)&&
    (fliplr(W2)K does not exist in dictionary) then
        L1← Length of characters in W1;
        L2← Length of characters in W2;
        L ← max(L1, L2);
        if (L == L1) then
            Output ← (Flag1, Index(W1), K);
            Add W1K in the dictionary;
        else
            Output ← (Flag2, Index(W2), K);
            Add W2K in the dictionary;
        end if
        W1 ← NIL;
        W2 ← NIL;
    end if
end while
```

Like LZ77, the maximum match is searched in both forward and reverse directions for all available dictionary entries. Once no further match is possible in either direction then a maximum match of characters in either direction is stored in L. The sent information contains the dictionary index, flag bit, and the next non-matching character in the incoming data stream. Again, the cost is an extra flag bit which indicates the dictionary reading direction.

C. Modified LZW84 Compression Algorithm

The original LZW84 algorithm is similar to the LZ78 algorithm in terms of a dictionary-based approach. However, for LZW84, there is an initialized dictionary containing the used characters (e.g. 256 ASCII characters) on both the compression and decompression sides. Additionally, the transmission of the not yet existing data symbol from the input data sequence is not done. This symbol is recovered from the first character of the dictionary entry which is transmitted next, since the algorithm moves on by a single character at a time. In case of a yet non-existing string, the first character of the previous sequence is appended to the previous sequence to obtain the missing entry.

The original algorithm is modified to include bi-directional reading alike the modifications in LZ77 and LZ78. However, the use of the flag bit for the initialized dictionary containing the ASCII characters is redundant. An optimal use of the flag bit for these entries can be made by exploiting the inherent bit symmetry of the original characters, e.g., the bit patterns of ASCII characters. By inverting each of the bits of the first 128 characters, the latter 128 characters can be obtained. Hence, the starting dictionary for the bi-directional approach has half the size than in its original counterpart. The original pseudo-code of LZW84 [3] is modified to include bi-directional reading in Algorithm 3.

In the modified algorithm the maximum match is searched in both forward and reverse directions through the dictionary. However, the last entry of the dictionary cannot be used for reverse reading due to the lag of one step which the decoder suffers in updating its dictionary. This necessary exclusion is illustrated in the following example where we allow reverse reading of the last entry on the encoder side. Let us assume that the input stream contains $ch1 + ch2 + ch1 + extra\ char$, and also that neither $ch1 + ch2$ nor $ch2 + ch1$ exists in the dictionary. The encoder reads in $ch1 + ch2$ and since no match is found, it appends this string as the last entry and transmits index($ch1$). The decoder receives index($ch1$), but cannot extend its dictionary since it has to wait for the first character of the next transmit. On the encoder side, the next string characteristically starts from the last unmatched character, hence, the relevant string becomes $ch2 + ch1$. A match in the dictionary is found at the last index in reverse direction, so the string is extended and becomes $ch2 + ch1 + extra\ char$, no match is found for this string. Hence, this string is appended and the transmitted code is index ($ch1 + ch2$) with a flag value of 1. On the decoder side, to update the last entry it required the first character of the current transmitted sequence which points towards a non-existing entry. Hence, the decoder cannot construct this dictionary entry since it cannot retrieve the relevant first character. This occurrence, although similar to the inherent exception of similar kind (i.e. index points to non-existing entry) in the original LZW84 algorithm, cannot be dealt with in a similar fashion. In the original case, the relevant character can always be recovered due to a certain repetition pattern. In the bi-directional case, however, it cannot due to the direction change. In our example, one would actually have two unknown characters, $ch2$ and $extra\ char$, that cannot both be

Algorithm 3 : Bi-directional LZW84

```
W1 = NIL (for forward matching);
W2 = NIL (for reverse matching);
prefix ← First input symbol;
nextbit ← perfix + 1;
codeword ← prefix + nextbit;
while input available do
  if codeword exists in dictionary then
    W1 ← length of codeword;
    ForwardMatch = 1;
  end if
  if codeword exists in flipped dictionary except last entry then
    W2 ← length of codeword;
    ReverseMatch = 1;
  end if
  if ForwardMatch = 1 or ReverseMatch = 1) then
    if (W1 > W2) then
      Flag = 0;
    else
      Flag = 1;
    end if
    W1 = 0;
    W2 = 0;
    nextbit = nextbit + 1;
    prefix ← codeword;
    ForwardMatch = 0;
    ReverseMatch = 0;
  end if
  if ForwardMatch ≠ 1 and ReverseMatch ≠ 1) then
    if flag = 0 then
      Output ← (Index(prefix), flag);
    else
      Output ← (index(prefix), flag);
      flag = 0;
    end if
    add codeword to dictionary;
    prefix ← last character of codeword;
    nextbit = nextbit + 1;
    codeword ← prefix + nextbit;
  end if
end while
```

resolved. Hence, we limit reverse reading to all but the last entry on the encoder side. In the appendix, we provide another example that flags the problem occurring when not restricting the last dictionary entry to forward comparison, only.

The information sent from the encoder to the decoder contains the dictionary index and the flag bit indicating the direction this entry is to be read. The cost of the modified algorithm is again this extra flag bit.

The bit-symmetry inversion has not been included in the above pseudo-code, since this particular manipulation does not alter the general structure of the modified algorithm. We need only note that in cases when the output (transmitted dictionary index) refers to the first 128 slots of the dictionary, the flag bit is to be interpreted as an indicator for bit inversion.

D. Simulation Results

For simulations, three different files were considered as examples for this paper, namely: a text file of an English novel having a size of 537 KB [8], an audio wave file of a drum sound having a size of 19 KB [9], and an image file of a spider having a size of 11,838 KB [10]. The results show that the modified algorithms perform better than the conventional ones, if there exists some symmetry in the data.

For LZ77, we have reserved the length of the lookahead buffer (L_{A}) to be 64 bytes. The number of bits required to address the search buffer are $N_{\mathrm{SB}} = \log_2(n - L_{\mathrm{A}})$. We have varied N_{SB} to observe its effect on the compression ratio for the actual and proposed LZ algorithms. Figure 4 shows the results for modified and original LZ77 algorithms. The compression ratio is computed as the ratio between the number of bits after compression to the original size. The results show that the bi-directional (FR) reading strategy performs better than the standard forward reading (F) in case of an image and audio file. However, for the text file, the two curves almost overlap in the start and later, the forward reading outperforms the bidirectional strategy highlighting the fact that less symmetry exists in the specified textual data.

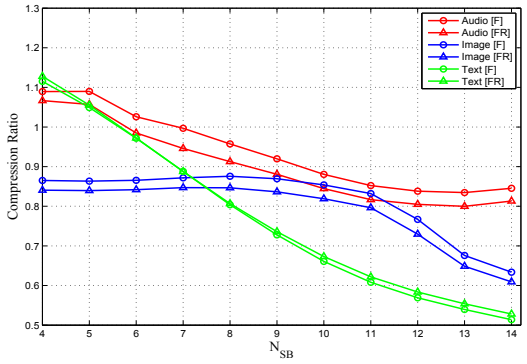

Fig. 4. Comparison of conventional and modified LZ77

For LZ78, we have varied the size of the dictionary using the number of bits required to address all locations of the dictionary (N_{D}). Figure 5 shows the corresponding results for the modified and the original LZ78 algorithm. The overall trend is the same with LZ77, with bi-directional reading yielding better compression ratios for lower N_{D}. However, note that as N_{D} increases, at some point, the original algorithm starts performing better than the modified one. For a very large dictionary size, the forward reading can provide as much compression as the bi-directional reading. The bi-directional variants, however, require an additional flag bit.

For LZW84, the dictionary size is varied in accordance with the number of bits required to address all locations of the dictionary. Figure 6 displays the relevant results. For the text file, the compression is better for small dictionary sizes, only. For the image and the audio file, however, bi-directional reading outperforms the original method.

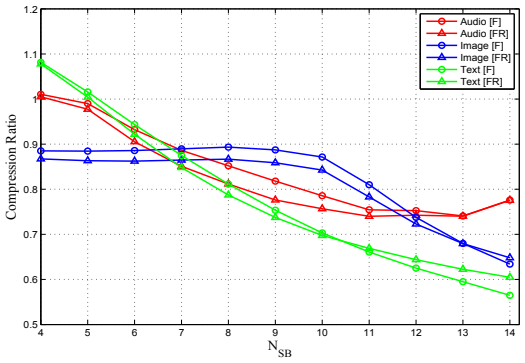

Fig. 5. Comparison of conventional and modified LZ78

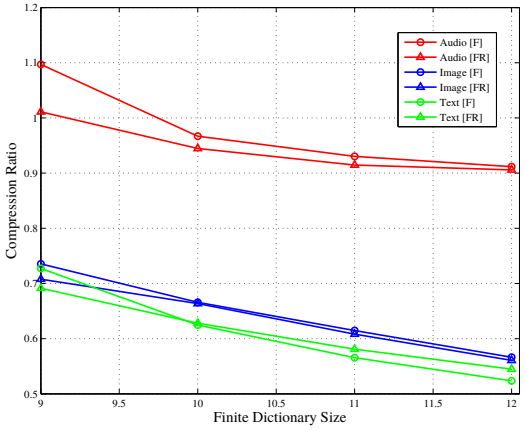

Fig. 6. Comparison of conventional and modified LZW84

IV. Conclusions

Motivated by bi-directional use of the DNA and alternative splicing this work has proposed modifications in LZ77, LZ78, and LZW84 algorithms for bi-directional reading. Results indicate that a better compression ratio can be achieved, if symmetries exist in the uncompressed data. Note that other symmetries can be utilized in a similar fashion, e.g., using conjugate reverse sequences in DNA compression or using more than two alternatives and hence, non-binary flags.

Appendix
Example for problem in bi-directional LZW84 when not restricting the last dictionary entry to forward reading, only

We assume that the dictionary only consists of the first 26 letters of the English alphabet in natural order. For simplicity, symmetry arising from bit pattern inversion is not considered. Let the input string be ⧫ ecabaced ⧫.

Encoder dictionary	Encoder Output	String at specified index	Decoder dictionary
27: ec	5,0	'e'	27: e?
28: ca	3,0	'c'	27: ec; 28: c?
29: ab	1,0	'a'	28: ca; 29: a?
30: bac	29,1	'?a'	29: a?; 30: ?a?
31: ced	27,1	'ce'	30: ?ac; 31: ce?

We observe that dictionary entries at indices 29 and 30 cannot be reconstructed by the decoder, i.e., in the end, 'b' is not resolved.

Acknowledgment

This work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG).

References

[1] Ziv, J., Lempel, A., "Compression of individual sequences via variable-rate coding," IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, 1977.
[2] Ziv, J., Lempel, A., "Capacities of equivalent channels in multilevel coding schemes," IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, 1978.
[3] Welch, T. A., "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8-19, 1984,
[4] M. Hoefnagels and M. Hoefnagels, *Biology: concepts and investigations*. McGraw-Hill Higher Education, 2009.
[5] T. Ogawa and T. Okazaki, "Discontinuous dna replication," *Annual review of biochemistry*, vol. 49, no. 1, pp. 421–457, 1980.
[6] S. Stamm, C. Smith, and R. Lührmann, *Alternative Pre-mRNA Splicing: Theory and Protocols*. John Wiley & Sons, 2012.
[7] Zeeh, C., "The Lempel-Ziv algorithm," January, 2003. [Online]. Available: http://tuxtina.de/les/seminar/LempelZiv.pdf
[8] Grimm, T. B., (2001, April) *Grimms' fairy tales*. [Online]. Available: http://www.gutenberg.org/cache/epub/2591/pg2591.txt
[9] (1998, September). http://www.stormii.com/Wavs/drum.wav
[10] (2007, December). http://compressionratings.com/files/cr_img2.tar.7z