Pre Equal Architecture for Lossless Data Compression and Decompression Using Hybrid Algorithm

Sandhya, Shravan Kulkarni, Yogesh Kini

SDMCET Dharwad

Abstract— In this paper, we propose a new two-stage algorithm that combines the features of both Lempel-Ziv-Welch (LZW) and Huffman coding algorithms. LZW is dictionary based algorithm, which is lossless in nature and incorporated as the standard of the consultative committee on International telegraphy and telephony [1]. Here, the code for each character is available in the dictionary which utilizes less number of bits (5 bits) than its ASCII code. The Huffman algorithm is most preferred as it has features like, more frequently occurred symbols will have shorter code words than the symbols that occur less frequently and the two symbols that occur least frequently will have the same length [1]. When the data is passed through these two stages it gets compressed to larger extent.

Index Terms— Huffman encoding algorithm, LZW compression algorithm, proposed two stage data compression and decompression algorithm.

I. INTRODUCTION

DATA compression is a method of encoding rules that allows substantial reduction in the total number of bits to store or transmit a file. Currently, two basic classes of data compression are applied in different areas [1]. One of these is lossy data compression, which is widely used to compress image data files for communication or archives purposes. The other is lossless data compression that is commonly used to transmit or archive text or binary files required to keep their information.

Lossless data compression algorithms mainly include Lempel and Ziv (LZ) codes [1], Huffman codes [3]. LZW algorithm is a lossless data compression algorithm which is based on dictionaries [1]. Each character is represented by an index number in the dictionary. Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream. In this paper, we proposed an improved scheme for data compression. When the input string passes through the LZW encoder, each character in dictionary is replaced with a code which is less number of bits than its ASCII code. This code is then passed through the Huffman encoder and thus the data gets compressed in two stages.

The block diagram of data compression model is described in Fig 1.

Fig 1: Data compression model [2]

II. RELATED WORK

Before presenting our new hardware architecture for data compression, we first discuss the features of the hardware implementations of the LZW and Huffman algorithms.

A. LZW Compression Algorithm

LZW compression algorithm is dictionary based algorithm which always output a code for a character. Each character has a code and index number in dictionary. Input data which we want to compress is read from file. Initially data is entered in buffer for searching in dictionary to generate its code. If there is no matching character found in dictionary. Then it will be entered as new character in dictionary and assign a code. If character is in dictionary then its code will be generate. Output codes have less number of bits than input data.
Algorithm: LZW Compression

**INPUT:** The data string to be compressed

**OUTPUT:** The encoded code words for compressed data

**Begin:**

1. **01:** Initialize table with single character strings
2. **02:** String = first input character
3. **03:** WHILE not end of input stream
4. **04:** Char = next input character
5. **05:** IF String + Char is in the string table
6. **06:** String = String + Char
7. **07:** ELSE
8. **08:** output the code for String
9. **09:** add String + Char to the string table
10. **10:** String = Char
11. **11:** END WHILE
12. **12:** output code for String

An example to show the data compression of LZW algorithm is given by,

Encode the string **BABAABRRRA** by the LZW encoding algorithm.

<table>
<thead>
<tr>
<th>Output</th>
<th>Code Word</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>256</td>
<td>BA</td>
</tr>
<tr>
<td>65</td>
<td>257</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>258</td>
<td>AAA</td>
</tr>
<tr>
<td>257</td>
<td>259</td>
<td>ABR</td>
</tr>
<tr>
<td>260</td>
<td>261</td>
<td>RRA</td>
</tr>
<tr>
<td>65</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

Fig 2. Example for LZW algorithm

The compressed message is: 66, 65, 256, 257, 82, 260, 65.

When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first codes going into the compressed file are simply the single bytes from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the decompression program to reconstruct the code table directly from the compressed data, without having to transmit the code table separately [1].

B. Huffman Algorithm.

Huffman Encoding Algorithms use the probability distribution of the alphabet of the source to develop the code words for symbols[1]. The frequency distribution of all the characters of the source is calculated in order to calculate the probability distribution. According to the probabilities, the code words are assigned. Shorter code words for higher probabilities and longer code words for smaller probabilities are assigned. For this task a binary tree is created using the symbols as leaves according to their probabilities and paths of those are taken as the code words.

Algorithm: Huffman Encoding

**INPUT:** The data string to be compressed.

**OUTPUT:** The encoded code words for input data

**Begin:**

1. **01:** n= length of input data stream
2. **02:** Calculate the probability of input symbols
3. **03:** WHILE not end of all data stream
4. **04:** IF (p[i]>p[i+1]) SWAP p[i] and p[i+1]
5. **05:** SWAP p[i] and p[i+1]
6. **06:** END WHILE
7. **07:** sum=p[n-1]+p[n]
8. **08:** WHILE size of n
9. **09:** IF (sum>p[i]) SWAP sum and p[i]
10. **10:** write the generated bits from right to left of individual probability and that is the output of Huffman algorithm.

An example to show the Huffman encoding is given by,

INPUT: **ABECADABEAC**

The tree implementation of Huffman encoding is given by,

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Probability</th>
<th>Code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>5/11</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2/11</td>
<td>110</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1/11</td>
<td>1110</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1/11</td>
<td>1111</td>
</tr>
</tbody>
</table>

Fig 3. Example for Huffman encoding

The tree implementation of Huffman encoding is given by,
In this paper, we propose a new two-stage algorithm that combines the features of both Lempel-Ziv-Welch (LZW) and Huffman coding algorithms. The input data can be still compressed if it is passed through two stages of compressions. We are combining the LZW and Huffman algorithms to increase the compression rate. The input data is first passed through LZW algorithm where we get the coded output for compressed data. Then it is converted to the binary form and probability of symbols are calculated. The calculated probability is arranged in decreasing order and given as input to the Huffman algorithm. Further Huffman encoding is done and the data get still compressed and thus higher compression rate is obtained.

Algorithm: Compression of data

1.01: Initialize table with string and the string with ‘$’
1.02: Initialize the dictionary
1.03: Convert a string array to cell array.
1.04: String = first input character
1.05: Read data stream until input Char not equal to ‘$’
1.06: IF String + Char is in the string table .
1.07: New dictionary = String + Char
1.08: ELSE
1.09: String=next char
1.10: END WHILE
1.11: a=length of lzw code words
1.12: Calculate probability of symbol.
1.13: WHILE
1.14: Temp=sort(set_prob)
1.15: Set_prob=last_prob+lastone_prob
1.16: End while

1.17: output code for String

Algorithm: Decompression of data

2.1: The original data is reconstructed i.e. decompression is done by using Huffman decoding in 1st step and then lzw.
2.2: Generate a tree equivalent to the encoding tree and then it is given to lzw decoding.
2.3: Create a loop which converts the cell array to char array.
2.4: Then finally we get the decompressed data.

II. ALGORITHM SIMULATION

To evaluate the compression performance of proposed algorithm we have done a matlab coding as per the algorithm specified above and simulation results are obtained. The simulation result is given by,

```
enter the string =’ABABABABABABABABCDABCDDDDDDCBABCDBACDADAS’
enter the dictionary =’ABCD’
```

dictnew =

Columns 1 through 10

Columns 11 through 20

Columns 21 through 25
‘CDB’ ‘BAC’ ‘CDBA’ ‘AC’ ‘CDA’
lzw code words

Columns 1 through 15
1 2 5 7 6 9 8 3 4 5 12 4 16 16 3

Columns 16 through 23
9 12 6 21 1 12 1 0

The allocated Huffman code words are:

2 3 3 11 12 13 14 4 5 15 4 0 10

The above box shows the simulation result of proposed two stage architecture.
Table I: Comparison of data compression at LZW and LZW+Huffman algorithms

<table>
<thead>
<tr>
<th>Input (Number of bytes)</th>
<th>LZW</th>
<th>LZW+Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Outpu t bytes</td>
<td>Compresse d bytes</td>
</tr>
<tr>
<td>42</td>
<td>29</td>
<td>13</td>
</tr>
<tr>
<td>60</td>
<td>46</td>
<td>14</td>
</tr>
<tr>
<td>80</td>
<td>53</td>
<td>27</td>
</tr>
</tbody>
</table>

Table II: Comparison of data compression ratios between LZW and LZW+Huffman algorithms

<table>
<thead>
<tr>
<th>Input (Number of bits)</th>
<th>Compression ratio at LZW algorithm</th>
<th>Compression ratio LZW+Huffman algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>30.95%</td>
<td>54.76%</td>
</tr>
<tr>
<td>60</td>
<td>23.33%</td>
<td>48.33%</td>
</tr>
<tr>
<td>80</td>
<td>33.75%</td>
<td>51.25%</td>
</tr>
</tbody>
</table>

IV. FUTURE SCOPE

The above two stage architecture can be realized to VLSI architecture which is faster in terms of processing with verilog coding.

The second stage of algorithm can be replaced with Adaptive-Huffman coding in linear way overcoming disadvantages of Huffman coding.

V. CONCLUSION

In this paper, we have presented a two structured architecture for data compression and decompression Technique. In the above architecture the compression rate is around 52% and compression ratio increases for larger number of input bits. We analyze compression rate with different number of input bits on matlab.

REFERENCES


◆◆◆