APPENDIX-A

RVLC CONSTRUCTION ALGORITHMS

Different construction algorithms to generate Reversible Variable Length Code are given below.

A.1. TAKISHIMA ALGORITHM TO GENERATE SYMMETRIC RVLCs [37]

This was the first algorithm designed for the construction of Reversible Variable Length code, developed by Takishima [37]. Before explaining the steps of the algorithm, let us discuss a few important terms related to this algorithm.

The number of symmetrical codewords on a full binary tree of level L is given by, \( m_0(L) = 2^{[L+1]/2} \), where \([x]\) is the floor function.

The prefix condition for allowing instantaneous decoding of the target symmetrical RVLC prohibits all of the symmetrical codewords in \( m_0(L) \) from serving as candidates, so all of the symmetrical codewords at level \( L \) that violate the prefix condition must be eliminated from \( m_0(L) \). Let \( m(L) \) denotes the number of available symmetrical RVLC codewords at level \( L \).

The steps of the algorithm are as follows:

**Step 1**: The bit length vector of the target symmetrical RVLC is initialized by bit length vector of Huffman code.

\[ n_{rev}(i) = n_h(i) \quad (\forall \ i \geq 1), \]

where \( n_{rev}(i) \) is the bit length vector of the target RVLC and \( n_h(i) \) is the bit length vector of Huffman Code. The number of codewords of length \( i \) in \( n_{rev}(i) \) is restricted by the bit alignment of codewords of length less than \( i \). The number of available codewords at any level \( i \) is defined by \( m(i) \).

**Step 2**: If \( n_{rev}(i) \leq m(i) \), \( n_{rev}(i) \) is unchanged, otherwise one bit length is added to some codewords, i.e.

\[ n_{rev}(i + 1) = n_{rev}(i + 1) + n_{rev}(i) - m(i) \text{ and } n_{rev}(i) = m(i) \]
Step 2 is repeated until the bit allocation is finished for all the codewords.

It may be noted that this algorithm has a problem of bit variation which encounters while selecting the codewords.

A.2. TSAI ALGORITHM TO GENERATE SYMMETRIC RVLCs [84]

Tsai symmetrical RVLC algorithm is in fact the modified version of the Takishima symmetrical RVLC algorithm. It solves the variation problem in Takishima algorithm and generates a more efficient symmetrical RVLC.

We first discuss some important terms related to this algorithm. Let $p(i)$ denote the total number of symmetrical codewords located at level $i$ which violate the prefix condition owing to some symmetrical codewords positioned in the path from the root to level $i$ that have been selected as target symmetrical codewords. As discussed in Takishima Symmetrical algorithm, $m_0(i)$ represents the total numbers of symmetrical codewords available at level $i$. Thus, the number of available symmetrical codewords, $m'(i)$, at level $i$ is

$$m'(i) = m_0(i) - p(i)$$

To simplify the algorithm, after the available candidate terms have been achieved they are arranged in the ascending order according to the maximum length of their symmetrical bit suffixes excluding the first bit of each candidate codeword. Now the selection procedure is carried forward. The steps of the algorithm are as follows:

**Step 1:** Initialize the bit-length vector of the target symmetrical RVLC, $n_{rev}(i)$, by the bit length vector $n_h(i)$ of the starting VLC i.e. the Huffman code.

**Step 2:** Calculate $m'(i)$ and the maximum length $b(i)$ of the symmetrical bit suffixes, excluding the first bit of each candidate codeword in $m'(i)$, and arrange the codewords in the increasing order of $a(i)$. If $n_{rev}(i) \leq m'(i)$, $n_{rev}(i)$ is unchanged. Else, the remaining $n_{rev}(i) - m'(i)$ codewords are added to $n_{rev}(i + 1)$ and assigned a symmetrical codeword for the next stage to make the candidate codewords as target codewords in the sequence, i.e.
$$n_{rev}(i+1) = n_{rev}(i+1) + n_{rev}(i) - m'(i) \text{ and } n_{rev}(i) = m'(i)$$

Step 3: Repeat Step 2 until every codeword has been assigned a Symmetrical RVLC codeword.

The problem of bit variation occurred in Takishima symmetrical RVLC algorithm approach is solved using this algorithm. This method gives coding efficiency improved to some extend as compared to Takishima symmetric RVLC algorithm, but not better than other algorithms.

A.3. **JEONG ALGORITHM TO GENERATE SYMMETRIC RVLCs [85]**

As compared to Takishima Symmetric RVLC algorithm and Tsai Symmetric RVLC algorithm, this algorithm is more simplified algorithm and reduces the complexity of the search method. The algorithm is based on the concept of generating half number of the required codewords on the half binary tree and then applying bit inversion to get the next set of remaining symmetrical codewords. It is also Huffman dependent code. Their dependence on Huffman code can be explained in terms of $Z_L$ adaptation. There are symmetrical codes words at every level that have all 0 bits. These are defined as $Z_L$ at level $L$.

The Algorithm steps can be summarized as:

**Step 1**: In the left half region of the binary Huffman tree, $Z_L$ is determined. The bit length of $Z_L$ is selected to be the same as that of the shortest Huffman code, $L_{\min}$.

**Step 2**: Until symmetrical codewords are selected as many as $S/2$, where $S$ is the number of codewords desired. All available symmetrical codewords are chosen from the highest level. The selection processes are followed by the elimination of codewords that violate the prefix condition at every level.

**Step 3**: Combining the already chosen symmetrical codewords and their bit-inversed versions codewords, a target symmetrical RVLC that can be instantaneously decoded is obtained.
This algorithm involves the generation of half binary tree and then the application of codeword search reduce the hardware complexity, since only half binary tree is required to be generated.

**A.4. YAN ALGORITHM TO GENERATE SYMMETRIC RVLCs [86]**

This algorithm is independent of Huffman codes. To explain this algorithm in detail, let the source symbols be represented by $S$. The entropy of the source, $H(S)$, is the theoretical limit of the average code length. Let $L_{\text{min}}$ be the minimum length of the RVLC codeword. To compute the minimum length of the codeword, $L_{\text{min}}$, $L_{\text{min},i}$ is used as a intermediated variable. The value of minimum codeword length, $L_{\text{min},i}$ can be calculated by

\[ L_{\text{min},i} = 1,2 \ldots \lfloor H(s) + 1 \rfloor \]

$L_{\text{min}}$ is assigned to that value of $L_{\text{min},i}$ which gives minimum value of average codeword length. It is obvious that $L_{\text{min}}$ should be less than $H(S)$ for a short average code length.

To generate the RVLC codewords for the source symbols, a basic type of container (or queue) is used as shown in Figure A.1. This container can be considered as a pipeline, in which according to the value of $L_{\text{min}}$, one binary number out of all possible binary numbers of length $L_{\text{min}}$ will be entered from the ‘en-queue’ side and after checking the condition for symmetry, the binary vector will either be selected as a codeword and de-queued from the ‘de-queue’ side or its derived codes (after padding ‘0’ and ‘1’ at the suffix side) will be entered from the ‘en-queue’ side as new candidates.

![Figure A.1 Pipeline structure, which is used as a queue to generate symmetric RVLCs](image)

In the process of selection, if a candidate code has been selected as a symmetrical Reversible Variable Length Codeword, all its sub tree candidate codes will not be pushed into the queue anymore. Only the derived Codes of those candidate codes which are asymmetrical will be
pushed into the queue as new candidate codes. That is to say, all of the candidate codes in the queue always satisfy the prefix condition. So what we have to do is to check whether the candidate code in the queue is symmetrical or not.

Let us consider a simple example, with the value of $L_{min}$ as 3, so the set of binary vectors of length 3 is \{000, 001, 010, 011, 100, 101, 110, 111\}. The first candidate, 000 is entered from the ‘en-queue’ side, and selected as a codeword due to being symmetric. Next vector is 001, as 001 is not symmetric so it is not considered as codeword, and it’s derived codes 0010 and 0011 are added to the list of binary vectors as new candidates. In this way, the list of binary vectors keeps on increasing due to derived codes. The next candidate, 010 is symmetric in nature, so it is considered a codeword. Similarly we check for the other binary numbers.

The basic steps of the algorithm are:

**Step 1:** Arrange the symbols in the decreasing order of their probabilities, and initialize the queue with all $2^{L_{min}}$ candidates of $L_{min}$ bits.

**Step 2:** De-queue the front candidate code, and check whether it is symmetrical. If so, select it as the target symmetrical Reversible Variable Length Code codeword and assign it to the successive symbol. If not, push its derived codes into the queue.

**Step 3:** Repeat Step 2 until every symbol has been assigned a symmetrical Reversible Variable Length Codeword.

### A.5. TAKISHIMA ALGORITHM TO GENERATE ASYMMETRIC RVLC [37]

This was the first algorithm proposed for asymmetric RVLC. An asymmetrical RVLC can be obtained from a base VLC (Huffman code) by adding bits. An instantaneously decodable VLC in the forward direction satisfies the prefix condition. To satisfy the additional condition of suffix-free, the suffix of each codeword can be extended by adding bits, which does not prevent instantaneous forward decoding. This property suggests an effective method of generating an asymmetrical RVLC. The steps to generate asymmetrical RVLC are as follows:
Step 1: The given symbols are arranged in the decreasing order of their occurrence probabilities and their corresponding Huffman Codes are generated.

Step 2: A binary tree is set up in such a way that the ends of the codewords are placed on the top from the shorter codewords to longer codewords.

Step 3: When a suffix which is also the prefix in the reverse tree coincides with the shorter codeword length, another codeword with the same bit length is assigned in the binary tree instead.

Step 4: When there is no other codeword with the same bit length whose suffix does not coincide with one of the shorter codewords, the minimum number of bits needed to satisfy the suffix condition are added to the of the codeword.

Step 5: After bit length assignment is completed, new codewords are sorted by bit length and they are reassigned to the symbols according to their occurrence probability.

A.6. **TSAI ALGORITHM TO GENERATE ASYMMETRIC RVLCs [127]**

This algorithm is also based on Huffman code. This algorithm has different criteria to add 0s and 1s in the codewords to satisfy the suffix conditions. It differs from the Takishima asymmetric algorithm in the way that it replaces the VLC obtained codeword by a shorter length available vector to satisfy suffix condition. The following are the steps for producing an asymmetrical RVLC:

Step 1: The symbols are arranged in the decreasing order of their occurrence probabilities and their corresponding Huffman Codes are generated.

Step 2: For every codeword, suffix condition will be checked. If a codeword is not satisfying the suffix condition, it will be replaced by other vector of minimum possible length.

Step 3: The steps will be repeated till we get the required number of codewords.
A.7. GOLOMB AND RICE CODE-ALGORITHM

The Golomb and Rice codes have recently been applied for coding of prediction errors in image coding applications. Golomb and Rice codes are nearly optimal for coding of exponentially distributed non-negative integers, and describe an integer \( n \) in terms of a quotient and a remainder. For simplicity, the divisor is often chosen to be a power of 2 i.e. \( 2^k \) and is parameterized by \( k \). The quotient can be arbitrarily large and is expressed using a unary representation; the remainder is bounded by the range \([0, (2^k - 1)]\) and is expressed in binary form using \( k \) bits. Depending on the parameter \( k \) taken, the Golomb Codes prefix and suffix length can be determined where the prefix consists of the quotient obtained by dividing the integer by \( 2^k \) and the remainder forms the suffix part.

Steps to obtain RVLCs are given as follows:

**Step 1:** Select a particular value of parameter \( k \). (The value of \( k \) is taken to be 1 throughout the implementation but can be changed if required).

**Step 2:** The prefix of the codeword is generated by dividing the integer value by \( 2^k \) and following the steps given below to generate the Golomb Code prefix:

1. If the length of the prefix is one (1), then the value of the prefix is taken to be ‘0’
2. If the length of the prefix is greater than 1, then the first and the last bit of the prefix is changed to 1 and the rest of the bits are made 0 without changing the length of the prefix

**Step 3:** The suffix for the Golomb codes is the remainder obtained by dividing integer by \( 2^k \).

**Step 4:** The above steps are repeated for all the symbols provided.

Golomb and Rice algorithm is nowadays being used in the video coding standards such as H.263++ and MPEG-4. The problem with Golomb and Rice algorithm is the increase in the codeword lengths with the number of codewords required.
APPENDIX-B

IMAGE COMPRESSION RESULTS

B.1. IMAGE COMPRESSION WITH DIFFERENT QUANTIZATION MATRICES

Test image - ‘Lena’

Figure B.1: Original image

Figure B.2: Decoded image with Q10

Figure B.3: Decoded image with Q50

Figure B.4: Decoded image with Q90

Test image - ‘Pepper’

Figure B.5: Original image

Figure B.6: Decoded image with Q10
Test image - ‘Airplane’

Figure B.7: Decoded image with Q50

Figure B.8: Decoded image with Q90

Figure B.9: Original image

Figure B.10: Decoded image with Q10

Figure B.11: Decoded image with Q50

Figure B.12: Decoded image with Q90
B.2. IMAGE COMPRESSION WITH DIFFERENT ENTROPY ENCODERS

Test image - ‘Lena’

Using Huffman code

Figure B.13: Original image

Figure B.14: Huffman decoded image

Using Reversible Variable Length Codes

Figure B.15: Original image

Figure B.16: Forward decoded image

Figure B.17: Reverse decoded image

Figure B.18: Combined image (RVLC decoded image)
Using Variable Length Error-correcting Codes

Figure B.19: Original image

Figure B.20: VLEC decoded image

Test image - ‘Pepper’

Using Huffman code

Figure B.21: Original image

Figure B.22: Huffman decoded image

Using Reversible Variable Length Codes

Figure B.23: Original image

Figure B.24: Forward decoded image
Using Variable Length Error-correcting Codes

Test image - ‘Airplane’

Using Huffman code
Using Reversible Variable Length Codes

Figure B.31: Original image
Figure B.32: Forward decoded image
Figure B.33: Reverse decoded image
Figure B.34: Combined Image (RVLC decoded image)

Using Variable Length Error-correcting Codes

Figure B.35: Original image
Figure B.36: VLEC decoded image
APPENDIX-C

VIDEO COMPRESSION RESULTS

Video-2 (Name: Pendulum)

Input Frames:

Decoded Frames:

Figure C.1: Frames of Video-2
Video-3 (Name: running man)

Input Frames:

Decoded Frames:

Figure C.2: Frames of Video-3
<table>
<thead>
<tr>
<th>Coding method</th>
<th>$L_{\text{min}}$</th>
<th>$L_{\text{max}}$</th>
<th>Total encoding bits</th>
<th>Average codeword length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huffman</td>
<td>2</td>
<td>17</td>
<td>61437</td>
<td>2.99</td>
</tr>
<tr>
<td>Yan RVLC</td>
<td>1</td>
<td>&gt;87</td>
<td>Very Large</td>
<td>&gt;4</td>
</tr>
<tr>
<td>2</td>
<td>87</td>
<td>Very Large</td>
<td>&gt;4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>67107</td>
<td>3.04</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>65925</td>
<td>3.19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>Very Large</td>
<td>&gt;5</td>
<td></td>
</tr>
<tr>
<td>Conventional H.263++ (Golomb and Rice RVLC)</td>
<td>1</td>
<td>28</td>
<td>84573</td>
<td>3.35</td>
</tr>
<tr>
<td>2</td>
<td>&gt;&gt;28</td>
<td>Very Large</td>
<td>&gt;4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>71685</td>
<td>3.16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>Very Large</td>
<td>&gt;6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>Very Large</td>
<td>&gt;7</td>
<td></td>
</tr>
</tbody>
</table>

Figure C.3: Comparison table of all entropy encoders for Video-2

<table>
<thead>
<tr>
<th>Coding method</th>
<th>$L_{\text{min}}$</th>
<th>$L_{\text{max}}$</th>
<th>Total encoding bits</th>
<th>Average codeword length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huffman</td>
<td>2</td>
<td>17</td>
<td>33845</td>
<td>2.99</td>
</tr>
<tr>
<td>Yan RVLC</td>
<td>1</td>
<td>&gt;87</td>
<td>Very Large</td>
<td>&gt;4</td>
</tr>
<tr>
<td>2</td>
<td>87</td>
<td>Very Large</td>
<td>&gt;4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>38110</td>
<td>3.04</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>39706</td>
<td>3.19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>&gt;39706</td>
<td>&gt;5</td>
<td></td>
</tr>
<tr>
<td>Conventional H.263++ (Golomb and Rice RVLC)</td>
<td>1</td>
<td>28</td>
<td>45187</td>
<td>3.35</td>
</tr>
<tr>
<td>2</td>
<td>&gt;&gt;28</td>
<td>&gt;45187</td>
<td>&gt;4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>41209</td>
<td>3.16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>&gt;41209</td>
<td>&gt;6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>&gt;41209</td>
<td>&gt;7</td>
<td></td>
</tr>
</tbody>
</table>

Figure C.4: Comparison table of all entropy encoders for Video-3