Assignment

ELL-786 Multimedia Systems

Submitted By:

Kartik Gupta (2018JTM2250)

&

Maj Vineet Sangwan (2019EET2594)

on

Implementation of an Efficient CABAC encoder

Bharti School Of

Telecommunication Technology and Management

IIT Delhi

India

June 28, 2020

Contents

List of Figures

1 CABAC

Context-adaptive binary arithmetic coding (CABAC) is a form of entropy encoding used in the H.264/MPEG-4 AVC and High Efficiency Video Coding (HEVC) standards. It is a lossless compression technique, although the video coding standards in which it is used are typically for lossy compression applications. CABAC is notable for providing much better compression than most other entropy encoding algorithms used in video encoding, and it is one of the key elements that provides the H.264/AVC encoding scheme with better compression capability than its predecessors.

CABAC has been adopted as a normative part of the H.264/AVC standard; it is one of two alternative methods of entropy coding in the new video coding standard. The other method specified in H.264/AVC is a low-complexity entropy-coding technique based on the usage of context-adaptively switched sets of variable-length codes, so-called Context-Adaptive Variable-Length Coding (CAVLC). Compared to CABAC, CAVLC offers reduced implementation costs at the price of lower compression efficiency. For TV signals in standard- or high-definition resolution, CABAC typically provides bit-rate savings of 10-20% relative to CAVLC at the same objective video quality.

2 Algorithm

The design of CABAC involves the key elements of binarization, context modeling, and binary arithmetic coding. These elements are illustrated as the main algorithmic building blocks of the CABAC encoding block diagram, as shown below.

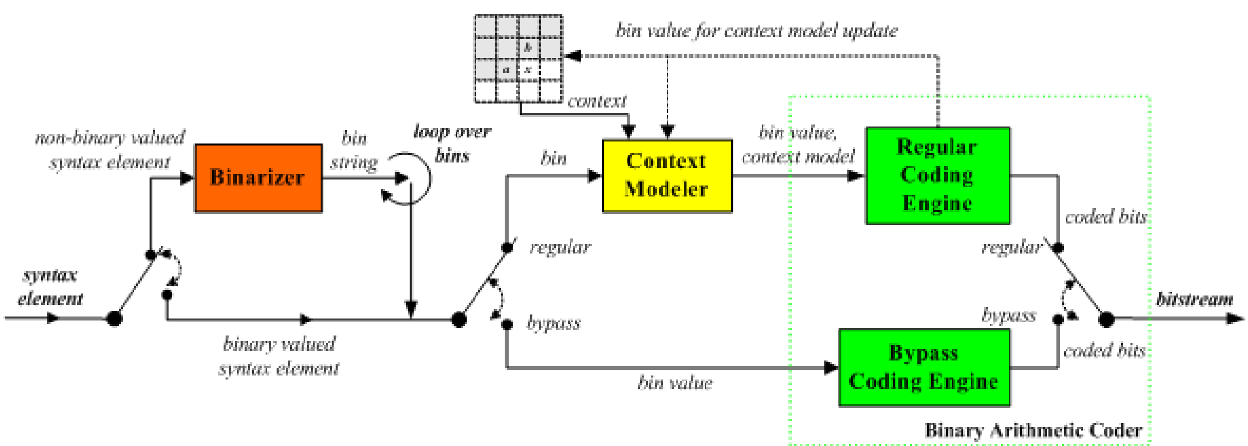

Figure 1: CABAC method of entropy encoding used within H264 video compression standard

- **Binarization :** The coding strategy of CABAC is based on the finding that a very efficient coding of syntax-element values in a hybrid block-based video coder, like components of motion vector differences or transform-coefficient level values, can be achieved by employing a binarization scheme as a kind of preprocessing unit for the subsequent stages of context modeling and binary arithmetic coding. In general, a binarization scheme defines a unique mapping of syntax element values to sequences of binary decisions, so-called bins, which can also be interpreted in terms of a binary code tree. The design of binarization schemes in CABAC is based on a few elementary prototypes whose structure enables simple online calculation and which are adapted to some suitable model-probability distributions.

- **Coding-Mode Decision and Context Modeling :** By decomposing each syntax element value into a sequence of bins, further processing of each bin value in CABAC depends on the associated coding-mode decision which can be either chosen as the regular or the bypass mode. The latter is chosen for bins related to the sign information or for lower significant bins which are assumed to be uniformly distributed and for which, consequently, the whole regular binary arithmetic encoding process is simply bypassed. In the regular coding mode, each bin value is encoded by using the regular binary arithmetic-coding engine, where the associated probability model is either determined by a fixed choice, without any context modeling, or adaptively chosen depending on the related context model. As an important design decision, the latter case is generally applied to the most frequently observed bins only, whereas the other, usually less frequently observed bins, will be treated using a joint, typically zero-order probability model. In this way, CABAC enables selective context modeling on a sub-symbol level, and hence, provides an efficient instrument for exploiting inter-symbol redundancies at significantly reduced overall modeling or learning costs. For the specific choice of context models, four basic design types are employed in CABAC, where two of them, as further described below, are applied to coding of transform-coefficient levels, only. The design of these four prototypes is based on a priori knowledge about the typical characteristics of the source data to be modeled and it reflects the aim to find a good compromise between the conflicting objectives of avoiding unnecessary modeling-cost overhead and exploiting the statistical dependencies to a large extent.

- **Pre-Coding of Transform-Coefficient Levels :** Coding of residual data in CABAC involves specifically designed syntax elements that are different from those used in the traditional run-length pre-coding approach. For each block with at least one nonzero quantized transform coefficient, a sequence of binary significance flags, indicating the position of significant (i.e., nonzero) coefficient levels within the scanning path, is produced in a first step. Interleaved with these significance flags, a sequence of so-called last flags (one for each significant coefficient level) is generated for signaling the position of the last significant level within the scanning path. This so-called significance information is transmitted as a preamble of the regarded transform block followed by the magnitude and sign information of nonzero levels in reverse scanning order. The context-modeling specifications for coding of binarized level magnitudes are based on the number of previously transmitted level magnitudes greater or equal to 1 within the reverse scanning path, which is motivated by the observation that levels with magnitude equal to 1 are statistical dominant at the end of the scanning path. For context-based coding of the significance information, each significance / last flag is conditioned on its position within the scanning path which can be interpreted as a frequency-dependent context modeling. Furthermore, for each of the different transforms (16x16, 4x4 and 2x2) in H.264/AVC (Version 1) as well as for luma and chroma component, a different set of contexts denoted as context category is employed. This allows the discrimination of statistically different sources with the result of a significantly better adaptation to the individual statistical characteristics.

- **Probability Estimation and Binary Arithmetic Coding :** On the lowest level of processing in CABAC, each bin value enters the binary arithmetic encoder, either in regular or bypass coding mode. For the latter, a fast branch of the coding engine with a considerably reduced complexity is used while for the former coding mode, encoding of the given bin value depends on the actual state of the associated adaptive probability model that is passed along with the bin value to the M coder - a term that has been chosen for the novel table-based binary arithmetic coding engine in CABAC. The specific features and the underlying design principles of the M coder can be found here. In the following, we will present some important aspects of probability estimation in CABAC that are not intimately tied to the M coder design.

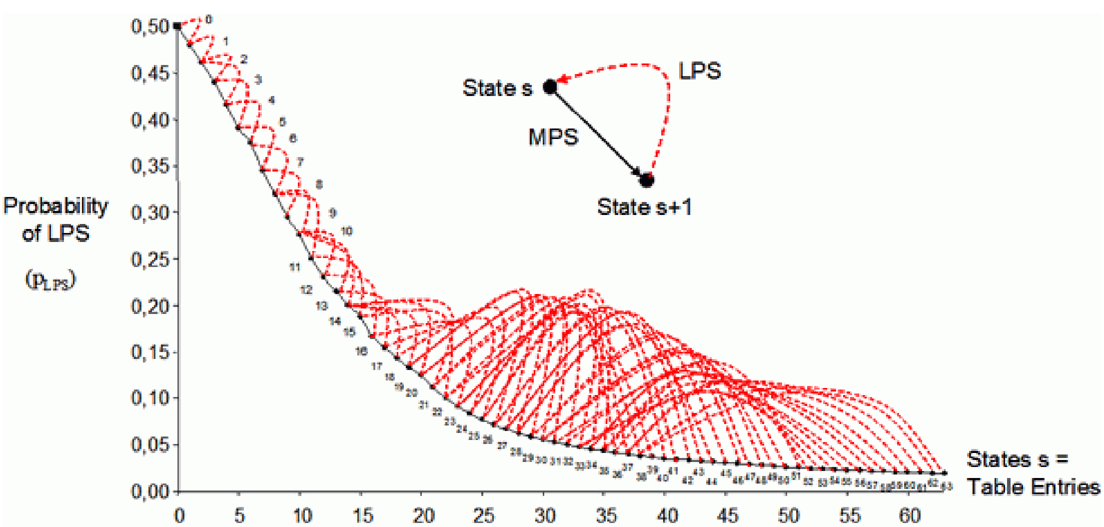

Figure 2: Plot

Probability estimation in CABAC is based on a table-driven estimator using a finite-state machine (FSM) approach with transition rules as illustrated above. Each probability model in CABAC can take one out of 128 different states with associated probability values p ranging in the interval $[0.01875, 0, 98125]$. Note that with the distinction between the least probable symbol (LPS) and the most probable symbol (MPS), it is sufficient to specify each state by means of the corresponding LPS-related probability $\mathrm{p_{LPS}} \in [0.01875, 0, 5]$. The method for designing the FSM transition rules was borrowed from Howaro and Vittere using a model of "exponential aging" with the following transition rule from time instance t to $t+1$

$$\mathrm{p}^{(t+1)}\mathrm{LPs} = \begin{cases} \mathrm{a} \cdot \mathrm{p}^{(\mathrm{t})} LPS & ifanMPSoccurs \\ \mathrm{a} \cdot \mathrm{p}^{(\mathrm{t})} LPS^{+}(1-\mathrm{a}), & ifanLPSoccurs \end{cases}$$

According to this relationship, the scaling factor α can be determined by $\mathrm{p_{min}} = 0.5\mathrm{a}^{\mathrm{N}-1}$, with the choice of $\mathrm{p_{min}} = 0.01875$ and $\mathrm{N} = 128/2 = 64$. Note however that the actual transition rules, as tabulated in CABAC and as shown in the graph above, were determined to be only approximately equal to those derived by this exponential aging rule.

Typically, without any prior knowledge of the statistical nature of the source, each model would be initialized with the state corresponding to a uniform distribution ($\mathrm{p} = 0.5$). However, in cases where the amount of data in the process of adapting to the true underlying statistics is comparably small, it is useful to provide some more appropriate initialization values for each probability model in order to better reflect its typically skewed nature. This is the purpose of the initialization process for context models in CABAC which operates on two levels. On the lower level, there is the quantization-parameter dependent initialization which is invoked at the beginning of each slice. It generates an initial state value depending on the given slicedependent quantization parameter SilceQP using a pair of so-called initialization parameters for each model which describes a modeled linear relationship between the SliceQP and the model probability p. As an extension of this low-level pre-adaptation of probability models, CABAC provides two additional pairs of initialization parameters for each model that is used in predictive (P) or bi-predictive (B) slices. since the encoder can choose between the corresponding three tables of initialization parameters and signal its choice to the decoder, an additional degree of pre-adaptation is achieved, especially in the case of using small slices at low to medium bit rates.

3 Problem Statement

The goal of this assignment is to implement an efficient CABAC (Context-adaptive binary arithmetic coding) encoder.You will implement a CABAC encoder and include as many proposed techniques as possible to improve its efficiency in terms of bit rate.

3.1 Implementation of Code

- **File Read :** The data files are read 1 byte at a time and converted into bit stream. C++ I/O (ifstream, ofstream) routines have been used for file reading/writing
- **Main Code :** The source code for CABAC is used which does the encoding given the 'context' and the 'current symbol'.
  Following routines (in red) of the source code provided are called:

```
FILE *fp;
fp=fopen("CABACencoded.dat", "w+"); //Stores the encoded data in this
//file
//calling the parameterized constructor for CABAC with the file pointer
QM obj(fp);

//Initialize the encoder parameters
obj.StartQM("encode");

//Encoding begins
WHILE Last Symbol read
      IF current symbol = 0
            gc = get Context based on previous n bits
            obj.encode(0,gc)
      ELSE IF current symbol = 1
            gc = get Context based on previous n bits
            obj.encode(0,gc)
      END IF
END WHILE

//Flush the remaining contents
obj.Flush();
```

- **File Write :** File Writing is done by the provided source code itself.

3.2 Preprocessing on Files

The preprocessing is done in the following way:

Step 1. The whole files is read and stored as bytes.
Step 2. First the MSB of each byte is read and stored, then the 2^{nd}MSB is read and stored and eventually the LSB is read and stored.

E.g. If the file contains 3 bytes $'140, 141, 150$ ' then the preprocessing is done as follows:

Bytes	140	141	150

Bit pattern	10001100	10001101	10010110

Step 1. Read the MSB

10001100 10001101 10010110

1 1 1

Step 2. Read the 2nd MSB

10001100 10001101 10010110

1 1 1 0 0 0

.

.

Step 8. Read the LSB

10001100 10001101 10010110

1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 0 1 0 1 0

Final bit stream is '111000000001110111001010'

This algorithm is chosen because:

- There is no drastic change between neighboring pixels/data in image/audio file. So the bytes are more or less same in neighboring areas.
- QM coder gives best compression when it gets a long stream of MPS.
- The above algorithm gives long stream of 0's and make the data amenable for compression using QM coding.

3.3 Results

The original files are read directly bit by bit and the CABAC encoder is used with Preprocessing on files for different values of context i.e. N = 2 , 3 , 4

		N = 2	
Input File	Original File Size (bytes)	Output File Size (bytes)	Compression Ratio (%)
text.dat	8358	6101	136.99
Image.dat	65536	57162	114.64
audio.dat	65536	45261	144.79
binary.dat	65536	7565	866.30

		N = 3	
Input File	Original File Size (bytes)	Output File Size (bytes)	Compression Ratio (%)
text.dat	8358	6107	136.85
Image.dat	65536	56985	115.00
audio.dat	65536	45146	145.16
binary.dat	65536	7505	873.23

		N = 4	
Input File	Original File Size (bytes)	Output File Size (bytes)	Compression Ratio (%)
text.dat	8358	6107	136.85
Image.dat	65536	56937	115.10
audio.dat	65536	45101	145.30
binary.dat	65536	7481	876.03

Figure 3: Compression Ratio for different context values (N)

All the Codes, Results and Input-Output files are available on GitHub. Click the below link :
https://github.com/2018JTM2250/Multimedia_Assignment3.git

4 Appendix

4.1 Main Code

```
//==============================================================================
// Name        : CABAC.cpp
// Description : CABAC main code
//==============================================================================


// including libraries
#include <iostream>
#include <math.h>
#include "QmCoder/qmcoder.h"
#include "FileIO/fileIO.h"
#include "PreProcessing/bitPlaneMap.h"


// defining functions
void printFileSize(int,char[], char[], int, int,int);
void shiftBit(bool *array, bool nextBit);
void printBoolArray(bool *array);
int getContext(bool *array);

using namespace std;

//Global Variables initialization
char *memblock;
int n;
int preprocess;

// Main program //
int main() {

  int flag1 = 0;
  int flag2 = 0;

  //User Interaction to enter file_name
  cout << "------------------------------" << endl;
  cout << "WELCOME" << endl;
  cout << "------------------------------" << endl;
  cout << "Enter the file to be encoded " << endl;
  char filename[1024];
```

```
  cin.getline(filename,1024);
    cout << "File has successfully uploaded" << endl;

  //User Interaction to enter the context
  while(flag1 == 0){
    cout << "—————————————————" << endl;
    cout << "Enter the number of context" << endl;
    cout << "You can select only 1,2,3,4" << endl;
    cin >> n;
    //Enter the # of context dependency
    if(n == 1 || n == 2 || n == 3 || n == 4 )
    {
      flag1 = 1;
    }
    else{
      cout << "Invalid Entry..!!!!" << endl;
      cout << "Enter either 1,2,3 or 4" << endl;
      cout << "—————————————————" << endl;
    }
  }

  //User interaction to do preprocessing

  while(flag2 == 0){
    cout << "—————————————————" << endl;
    cout << "Select the option by entering 1 or 2: "<<endl;
    cout << "1. Do bitMap Preprocessing " << endl;
    cout << "2. Don't do Preprocessing " << endl;
    cin >> preprocess;

    if(preprocess == 1 || preprocess == 2)
    {
      cout << "You have chosen option : " << preprocess << endl;
      flag2 = 1;
    }
    else{
      cout << "Invalid Entry..!!!!" << endl;
      cout << "Enter either 1 or 2" << endl;
      cout << "—————————————————" << endl;
    }
  }


//——————————————————————————————————————————————————//
//——————Main code begins————————————————//

  //Array to store the previous (memory) for context bits
  bool *array = new bool[n];
```

```
  int i=0;
  while (i < n)
  {
    array[i] = 0;
    i++;
  }

  // Read the Input file
  //  char filename[] = "resources/image.dat";
  //read the contents of the file and store it in memblock
  memblock = readFileByBytes(filename);
  int originalFileSize = FileSizeinBytes;

  //Do PreProcessing
  //This will store the preprocessed file in *PreProcessed.dat* file
  if(preprocess == 1){
    bitmapProcessing(memblock,originalFileSize);
    //Pre-processed Input filename
    char filename1[] = "PreProcessed.dat";
    //read the contents of the file and store it in memblock
    memblock = readFileByBytes(filename1);
  }

  //output the data to this file
  FILE *fp;
  fp=fopen("CABACencoded.dat", "w+");
  //calling the parameterized constructor for CABAC
 QM obj(fp);
  //Initialize the encoder parameters
  obj.StartQM("encode");

  //Read the file till EOF
  while (!checkEOF()) {
    if (ReadBit() == 0){
      int gc =  getContext(array);
      obj.encode(0,getContext(array));
      shiftBit(array,0);
    }
    else{
      int gc = getContext(array);
      obj.encode(1,getContext(array));
      shiftBit(array,1);
    }
  }

  //Flush the remaining contents
  obj.Flush();
  cout << "Done Encoding :)";
  fclose(fp);

```

```
  //Output file Size
  char outputFilename[] = "CABACencoded.dat";
  char *opFilename = outputFilename;
  int encodedFileSize = getFileSize(opFilename);

  //Print the result
  printFileSize(n,filename,outputFilename,encodedFileSize,originalFileSize,
    preprocess);
//
//  cout << "Done/!!";
  //return 0;
}






/**
 * This function changes the context when a
 * new symbol is read.
 * Basically it shifts the array to the right.
 * In the bool array, position 0 is the most recent position.
 */


void shiftBit(bool *array, bool nextBit){
  int i = n-1;
  while (i >= 0)
  {
    if(i == 0){
      array[i] = nextBit;
    }

    else{
      array[i] = array[i-1];
    }

  i = i-1;
  }
}


void printBoolArray(bool *array){
  int i = 0;
  while (i < n)
  {
    cout << array[i] << " ";
    i++ ;
  }
```

```
  cout << endl;
}


/**
 * THis function computes the context based
 * on the bool array.
 * e.g. if the bool array has entries {0,1,1},
 * then the context is 2^(0)*0 + 2^(1)*1 + 2^(2)*1 = 5
 */
int getContext(bool *array){
  int context = 0;
  int i = 0;

  while (i < n)
  {
    context += pow(2,i)*array[i];
    i++;
  }

  return context;
}

/**
 * This function prints the input and output info.
 */
void printFileSize(int n, char inputFilename[], char outputFilename[], int encodedFileSize, int originalFileSize, int preprocess){

  // Displaying input information
  cout << "--------------------------------------------" << endl;
  cout << "INPUT Information" << endl ;
  cout << "--------------------------------------------" << endl ;


  cout << "Input File Name      : " <<inputFilename << endl;
  cout << "Context Dependency   : " << n << endl;

  if(preprocess == 1){
    cout << "PreProcessing :  YES" << endl;
  }
  else{
    cout << "PreProcessing : NO " << endl;
  }

  cout << "Original File size   : " << originalFileSize << " bytes" << endl;


  // Displaying output information
  cout << "--------------------------------------------" << endl;
```

```
  cout << "OUTPUT Information" << endl ;
  cout << "--------------------------------------------" << endl;


  cout << "Output File Name      : " << outputFilename << endl;
  cout << "Compressed File size : " << encodedFileSize << " bytes" << endl;
  cout << "Compression Ratio     : " << (float)encodedFileSize/originalFileSize *100 << "%" << endl << endl;
  cout << "---------------------------------------------" << endl;
}
```

4.2 Input and Output File Code

```
//============================================================================
// Name        : fileIO.cpp
// Description : Reading and Writing files
//============================================================================


// including header file "fileIO.h"
#include "fileIO.h"

// declaring a global variable for storing the size of the file in Bytes
int FileSizeinBytes;

// Stream class to write on files
std::ofstream myFile;

using namespace std;

// function to get size of the file
int getFileSize(char *filename){

  streampos size;  // stream position type

  //Stream class to read from files
  // ios::in - open a file for reading
  // ios::ate-Open a file for output and move the read/write control to the end of the file.
  ifstream file (filename, ios::in|ios::binary|ios::ate);

  if(file.is_open()) {
    //cout << "Reading size of the file" << endl;
      size = file.tellg();
      file.close();
  }
  else{
    cout << "opening desired file for getting size of the file is unsuccessful" << endl;
  }
```

```
  return size;
}

// function to reaf content of the file
char* readFileByBytes(char* fileName){

    streampos size;   // stream position type
    char * memblock;

    //Stream class to read from files
    // ios::in - open a file for reading
    // ios::ate-Open a file for output and move the read/write control to the end of the file.
     ifstream file (fileName, ios::in|ios::binary|ios::ate);

    if (file.is_open())
      {
      //cout << "Reading data of the file" << endl;
        size = file.tellg();   // returns the current     get     position of the pointer in the stream
        memblock = new char [size];
        file.seekg (0, ios::beg);
        file.read (memblock, size);
        file.close();

        cout << "File content has successfully placed in memory" <<endl;
        FileSizeinBytes = size;

      }

      else {
        cout << "Opening file for reading content is unsuccessful" << endl ;
      }

      return memblock;
}

void WriteByte(unsigned char byte){
  cout << " " << (int)byte ;
}


/**
 * write the file.
 * @param -
 * 1. char data : data you want to write in the file.
 */
void writeFileByBytes(unsigned char data){
  //cout << "data written in file" << endl;
  myFile << data;
```

```
}

namespace Wr{
  unsigned char b;
  int s;
}

/**
 * This function write the bits given from MSB-> LSB
 * The first bit is written to the MSB, and so on the
 * 8th bit written to the LSB.
 */

void WriteBit(bool x)
{
    Wr::b |= (x ? 1 : 0) << (7-Wr::s);
    Wr::s++;

    if (Wr::s == 8)
    {
      writeFileByBytes(Wr::b);
        Wr::b = 0;
        Wr::s = 0;
    }
}


void writeSingleCode(unsigned long code, char size) {

  for(int i = 0; i < size; i++){
        bool x = ((code & (1 << i))?1:0);
          WriteBit(x);
    }
}

/**
 * Check the last bits status
 */
void checkStatusOfLastBit(){
  if(Wr::s <= 8){
    for (int i = 0; i < Wr::s; i++)
      Wr::b |= 0 << Wr::s;
    writeFileByBytes(Wr::b);
  }
}

/**
 * Open the file once.
 * @param-
 * 1. fileName : name of the file in which you
```

```
 * want to write
 */
void writePrepare(char *fileName){
    //ios::app-> append to end of file.//No need to append
    //ios::binary-> file is binary not text.
    //ios::out -> write to the file
    myFile.open(fileName, ios::out|ios::binary);
}

//for read bits
namespace RB {
  int pointer;
  unsigned char b1;
  int s1;
}

/**
 * This function reads the file bit by bit (starting from MSB)
 * with the help of 3 global variables.
 * pointer -> stores the current location in the memory block.
 * b1 -> stores the current symbol.
 * s1 -> stores the current count from the 8 bits.
 */
bool ReadBit() {
  if (RB::s1 == 0) {
    RB::b1 = memblock[RB::pointer++];
//    cout << (char)b1 << endl;
    RB::s1 = 8;
  }
//  cout << "s: " << RB::s1 << endl;

  bool bit = (RB::b1 >> (RB::s1-1)) & 1;
  RB::s1--;
  return bit;
}

/**
 * This function supports the read function and helps
 * find out the EOF
 */
bool checkEOF() {
  bool bit = 0;
  //IF the count is equal to FileSize and s1-> points to LSB.
  if (RB::pointer == (FileSizeinBytes) && RB::s1 == 0)
    bit = 1;
//  cout << "pointer: " << RB::pointer << endl;
  return bit;
}


```

```


/**
 * Close the file once all write operations done.
 */
void closeFile(){
  myFile.close();
}
```

4.3 Code for Preprocessing

```
//============================================================================
// Name        : bitPlaneMap.cpp
// Description : Bitmap Preprocessing
//============================================================================



#include "bitPlaneMap.h"


/**
 * This function creates a mask for reading bits
 * starting from MSB -> LSB
 */
void createMask(vector <int> &mask){
  mask.push_back(0x80);
  mask.push_back(0x40);
  mask.push_back(0x20);
  mask.push_back(0x10);
  mask.push_back(0x08);
  mask.push_back(0x04);
  mask.push_back(0x02);
  mask.push_back(0x01);
}


/**
 * This function does the bitmap processing.
 * Read MSB's of all the 8-bit blocks, store it as stream,
 * then read 2nd MSB, store it in stream and so on.
 * e.g.
 * If the memory block has 2 elements:
 * memblock[0] = 01100010;
 * memblock[1] = 01110001;
 *
 * output bit stream: 0011110100001010
 */
void preprocessFile(char *memblock, int fileSize){

```

```
  vector <int> mask;
  createMask(mask);

  for(int i = 0; i < 8; i++){
      for(int j = 0; j < fileSize; j++){
        if((memblock[j] & mask.at(i)) != 0){
//            cout << 1 << " ";
          WriteBit(1);
        }
        else{
//            cout << 0 <<" ";
          WriteBit(0);
          }
        }//end of main if

      }
cout << endl;
}


/**
 * This function takes the input data read, does the bitmap
 * processing and stores the result in *PreProcessed.dat* file.
 * @param:
 * 1. char *memblock - Memory read
 * 2. int filesize - size of the input file in bytes
 */

void bitmapProcessing(char *memblock, int fileSize){
  //Write prepare the output file.
  char opFile[] = "PreProcessed.dat";
  char *outputFile = opFile;
  writePrepare(outputFile);

  //pre-process the file
  preprocessFile(memblock, fileSize);
  cout << "Preprocessed File stored in *PreProcessed.dat*" << endl;
  closeFile();

}
```

4.4 Code for QM coder

```
//============================================================================
// Name        : qmcoder.cpp
// Description : QM coder
//============================================================================



```

```
#include <stdlib.h>
#include "qmcoder.h"

#define QMputc(BP, m_File)    if (bFirst) {fputc(BP, m_File);} else {bFirst = 1;};

using namespace std;

int lsz[256]= {
  0x5a1d, 0x2586, 0x1114, 0x080b, 0x03d8,
  0x01da, 0x0015, 0x006f, 0x0036, 0x001a,
  0x000d, 0x0006, 0x0003, 0x0001, 0x5a7f,
  0x3f25, 0x2cf2, 0x207c, 0x17b9, 0x1182,
  0x0cef, 0x09a1, 0x072f, 0x055c, 0x0406,
  0x0303, 0x0240, 0x01b1, 0x0144, 0x00f5,
  0x00b7, 0x008a, 0x0068, 0x004e, 0x003b,
  0x002c, 0x5ae1, 0x484c, 0x3a0d, 0x2ef1,
  0x261f, 0x1f33, 0x19a8, 0x1518, 0x1177,
  0x0e74, 0x0bfb, 0x09f8, 0x0861, 0x0706,
  0x05cd, 0x04de, 0x040f, 0x0363, 0x02d4,
  0x025c, 0x01f8, 0x01a4, 0x0160, 0x0125,
  0x00f6, 0x00cb, 0x00ab, 0x008f, 0x5b12,
  0x4d04, 0x412c, 0x37d8, 0x2fe8, 0x293c,
  0x2379, 0x1edf, 0x1aa9, 0x174e, 0x1424,
  0x119c, 0x0f6b, 0x0d51, 0x0bb6, 0x0a40,
  0x5832, 0x4d1c, 0x438e, 0x3bdd, 0x34ee,
  0x2eae, 0x299a, 0x2516, 0x5570, 0x4ca9,
  0x44d9, 0x3e22, 0x3824, 0x32b4, 0x2e17,
  0x56a8, 0x4f46, 0x47e5, 0x41cf, 0x3c3d,
  0x375e, 0x5231, 0x4c0f, 0x4639, 0x415e,
  0x5627, 0x50e7, 0x4b85, 0x5597, 0x504f,
  0x5a10, 0x5522, 0x59eb
};

int nlps[256]= {
  1, 14, 16, 18, 20,   23, 25, 28, 30, 33,
  35,  9, 10, 12, 15,   36, 38, 39, 40, 42,
  43, 45, 46, 48, 49,   51, 52, 54, 56, 57,
  59, 60, 62, 63, 32,   33, 37, 64, 65, 67,
  68, 69, 70, 72, 73,   74, 75, 77, 78, 79,
  48, 50, 50, 51, 52,   53, 54, 55, 56, 57,
  58, 59, 61, 61, 65,   80, 81, 82, 83, 84,
  86, 87, 87, 72, 72,   74, 74, 75, 77, 77,
  80, 88, 89, 90, 91,   92, 93, 86, 88, 95,
  96, 97, 99, 99, 93,   95,101,102,103,104,
  99,105,106,107,103,  105,108,109,110,111,
  110,112,112
};
```

```
int nmps[256]= {
  1,  2,  3,  4,  5,    6,  7,  8,  9, 10,
  11, 12, 13, 13, 15,   16, 17, 18, 19, 20,
  21, 22, 23, 24, 25,   26, 27, 28, 29, 30,
  31, 32, 33, 34, 35,    9, 37, 38, 39, 40,
  41, 42, 43, 44, 45,   46, 47, 48, 49, 50,
  51, 52, 53, 54, 55,   56, 57, 58, 59, 60,
  61, 62, 63, 32, 65,   66, 67, 68, 69, 70,
  71, 72, 73, 74, 75,   76, 77, 78, 79, 48,
  81, 82, 83, 84, 85,   86, 87, 71, 89, 90,
  91, 92, 93, 94, 86,   96, 97, 98, 99,100,
  93,102,103,104, 99,  106,107,103,109,107,
  111,109,111
};

int swit[256]= {
  1,0,0,0,0,    0,0,0,0,0,
  0,0,0,0,1,    0,0,0,0,0,
  0,0,0,0,0,    0,0,0,0,0,
  0,0,0,0,0,    0,1,0,0,0,
  0,0,0,0,0,    0,0,0,0,0,
  0,0,0,0,0,    0,0,0,0,0,
  0,0,0,0,1,    0,0,0,0,0,
  0,0,0,0,0,    0,0,0,0,0,
  1,0,0,0,0,    0,0,0,1,0,
  0,0,0,0,0,    1,0,0,0,0,
  0,0,0,0,0,    1,0,0,0,0,
  1,0,1
};

QM::QM(FILE *FP)
{
  m_File = FP;
  max_context = 4096;
  //max_context = 3;
  st_table = (unsigned char *) calloc(max_context, sizeof(unsigned char));
  mps_table= (unsigned char *) calloc(max_context, sizeof(unsigned char));
}

void QM::StartQM(const char *direction)
{
  if (! strcmp(direction, "encode"))
  {
    sc = 0;
    A_interval = 0x10000;
    C_register = 0;
    ct = 11;

    count = -1;
    debug = 0;
```

```
    BP = 0;
    bFirst = 0;
  }
  else if(! strcmp(direction, "decode"))
  {
    count = 0;

    MPS = 0;
    A_interval = 0x10000 ;
    C_register = 0 ;
    bEnd = 0;

    Byte_in( );
    C_register <<= 8 ;
    Byte_in( );
    C_register <<= 8 ;
    Cx = (unsigned) ((C_register & 0xffff0000) >> 16)  ;

    ct = 0;
    debug = 0;
  }
  else{
    cout << "Command " << direction << " cannot be recognized, please use encode/decode only." << endl;
  }
}


QM::~QM()
{
  free(st_table);
  free(mps_table);
}


void
QM::reset()
{
  for (int i = 0; i < max_context; i++)
  {
    st_table[i] = 0;
    mps_table[i] = 0;
  }
}


void
QM::encode(unsigned char symbol, int context )
{
  if (this->debug) cout <<(char) (symbol+'0') << " " << context << endl;
```

```

  if (context >= max_context)
  {
    unsigned char *new_st, *new_mps;
    new_st = (unsigned char *) calloc(max_context*2, sizeof(unsigned char));
    new_mps= (unsigned char *) calloc(max_context*2, sizeof(unsigned char));
    memcpy(new_st, st_table, max_context*sizeof(unsigned char));
    memcpy(new_mps,mps_table,max_context*sizeof(unsigned char));
    max_context *= 2;
    free(st_table);
    free(mps_table);
    st_table = new_st;
    mps_table = new_mps;
  }

  next_st = cur_st = st_table[context];
  next_MPS = MPS = mps_table[context];
  Qe  = lsz[ st_table[context] ];

  if (MPS == symbol)
    Code_MPS( );
  else
    Code_LPS( );

  st_table[context] = next_st;
  mps_table[context] = next_MPS;
};


void
QM::encode(unsigned char symbol, int prob, int mps_symbol )
{
  if (this->debug) cout <<(char) (symbol+'0') << " " << prob << endl;

  next_st = cur_st = 0;
  next_MPS = MPS = mps_symbol;
  Qe  = prob;

  if (MPS == symbol)
    Code_MPS();
  else
    Code_LPS();
};


void
QM::Flush()
{
  Clear_final_bits();
  C_register <<= ct ;
```

```
  Byte_out();
  C_register <<= 8;
  Byte_out();
  QMputc(BP, m_File);
  QMputc(0xff, m_File);count++;
  QMputc(0xff, m_File);count++;
}


unsigned char
QM::decode(int context)
{
  if (context >= max_context)
  {
    unsigned char *new_st, *new_mps;
    new_st = (unsigned char *) calloc(max_context*2, sizeof(unsigned char));
    new_mps= (unsigned char *) calloc(max_context*2, sizeof(unsigned char));
    memcpy(new_st, st_table, max_context*sizeof(unsigned char));
    memcpy(new_mps,mps_table,max_context*sizeof(unsigned char));
    max_context *= 2;
    free(st_table);
    free(mps_table);
    st_table = new_st;
    mps_table = new_mps;
  }
  next_st = cur_st = st_table[context];
  next_MPS = MPS = mps_table[context];
  Qe  = lsz[ st_table[context] ];
  unsigned char ret_val = AM_decode_Symbol();
  st_table[context] = next_st;
  mps_table[context] = next_MPS;

  if (this->debug) cout <<(char) (ret_val+'0') << " " << context << endl;
  return ret_val;
};


unsigned char
QM::decode(int prob, int mps_symbol)
{
  next_st = cur_st = 0;
  next_MPS = MPS = mps_symbol;
  Qe = prob;
  unsigned char ret_val = AM_decode_Symbol();

  if (this->debug) cout <<(char) (ret_val+'0') << " " << prob << endl;
  return ret_val;
};
```

```
void
QM::Code_LPS()
{
  A_interval -= Qe ;

  if (!(A_interval < Qe))
  {
    C_register += A_interval ;
    A_interval = Qe ;
  }

  if (swit[cur_st] == 1)
  {
    next_MPS = 1 - MPS;
  }
  next_st = nlps[cur_st];

  Renorm_e();
};


void
QM::Code_MPS()
{
  A_interval -= Qe ;

  if (A_interval < 0x8000)
  {
    if (A_interval < Qe)
    {
      C_register += A_interval ;
      A_interval = Qe ;
    }
    next_st = nmps[cur_st];
    Renorm_e();
  }
}


void
QM::Renorm_e()
{
  while (A_interval < 0x8000)
  {
    A_interval <<= 1 ;
    C_register <<= 1 ;
    ct--;

    if (ct == 0)
    {
```

```
      Byte_out();
      ct = 8 ;
    }
  }
}


void
QM::Byte_out()
{
  unsigned t = C_register>>19;

  if (t > 0xff)
  {
    BP++;
    Stuff_0();
    Output_stacked_zeros();
    QMputc(BP, m_File);count++;
    BP = t;
  }
  else
  {
    if (t == 0xff)
    {
      sc++;
    }
    else
    {
      Output_stacked_0xffs();
      QMputc(BP, m_File);count++;
      BP = t;
    }
  }
  C_register &= 0x7ffff;
}


void
QM::Output_stacked_zeros()
{
  while (sc > 0)
  {
    QMputc(BP, m_File);count++;
    BP = 0;
    sc--;
  }
}


void
```

```
QM::Output_stacked_0xffs()
{
  while (sc > 0)
  {
    QMputc(BP, m_File);count++;
    BP = 0xff ;
    QMputc(BP, m_File);count++;
    BP = 0 ;
    sc--;
  }
}


void
QM::Stuff_0()
{
  if (BP == 0xff)
  {
    QMputc(BP, m_File);count++;
    BP = 0;
  }
}


void
QM::Clear_final_bits()
{
  unsigned long t;
  t = C_register + A_interval - 1 ;
  t &= 0xffff0000 ;

  if (t < C_register) t += 0x8000 ;

  C_register = t ;
}


unsigned char
QM::AM_decode_Symbol()
{
  unsigned char D;

  A_interval -= Qe ;

  if (Cx < A_interval)
  {
    if (A_interval < 0x8000)
    {
      D = Cond_MPS_exchange();
      Renorm_d();
```

```
    }
    else
      D = MPS;
  }
  else
  {
    D = Cond_LPS_exchange();
    Renorm_d();
  }

  return D;
}


unsigned char
QM::Cond_LPS_exchange()
{
  unsigned char D;
  unsigned  C_low;


  if (A_interval < Qe)
  {
    D = MPS;
    Cx -= A_interval;
    C_low = C_register & 0x0000ffff;

    C_register = ((unsigned long)Cx<<16) + (unsigned long)C_low;
    A_interval = Qe;
    next_st = nmps[ cur_st ];
  }
  else
  {
    D = 1 - MPS;
    Cx -= A_interval;
    C_low = C_register & 0x0000ffff;
    C_register = ((unsigned long)Cx << 16) + (unsigned long)C_low;
    A_interval = Qe;

    if ( swit[ cur_st ]==1 )
    {
      next_MPS = 1-MPS;
    }
    next_st = nlps[ cur_st ];
  }

  return D;
}


```

```
unsigned char
QM::Cond_MPS_exchange( )
{
  unsigned char D;

  if (A_interval < Qe)
  {
    D = 1 - MPS;
    if (swit[cur_st] == 1)
    {
      next_MPS = 1 - MPS;
    }
    next_st = nlps[cur_st];
  }
  else
  {
    D = MPS;
    next_st = nmps[cur_st];
  }

  return D;
}


void
QM::Renorm_d( )
{
  while (A_interval<0x8000)
  {
    if (ct==0)
    {
      if (bEnd == 0) Byte_in();
      ct = 8 ;
    }
    A_interval <<= 1 ;
    C_register <<= 1 ;
    ct--;
  }

  Cx = (unsigned) ((C_register & 0xffff0000) >> 16);
};


void
QM::Byte_in( )
{
  unsigned char B;
  B = fgetc(m_File), count++;

  if (B == 0xff)
```

```
    {
        Unstuff_0();
    }
    else
    {
        C_register += (unsigned) B << 8 ;
    }
};


void
QM::Unstuff_0( )
{
    unsigned char B;
    B = fgetc(m_File), count++;

    if( B == 0 )
    {
        C_register |= 0xff00 ;
    }
    else
    {
        if( B == 0xff )
        {
            //cerr << "\nEnd marker has been met!\n";
            bEnd = 1;
        }
    }
}


int QM::Counting()
{
    if (ct == 0)
    {
        return count*8;
    }
    else
    {
        return count*8+8-ct;
    }
}
```

References

[1] Jigar. *CABAC*. GitHub. https://github.com/jigar23/CABAC.

[2] J.D. Bruguera R.R. Osorio. *Arithmetic coding architecture for H.264/AVC CABAC compression system*. IEEE. https://ieeexplore.ieee.org/document/1333259.

[3] VCodex. *H.264/AVC Context Adaptive Binary Arithmetic Coding (CABAC)*. https://www.vcodex.com/h264avc-context-adaptive-binary-arithmetic-coding-cabac/.

[4] Wikipedia. *Context-adaptive binary arithmetic coding*. https://en.m.wikipedia.org/wiki/Context-adaptive_binary_arithmetic_coding.