**Image Encryption Compression Documentation**

**Team No. : 16**

**Team members:-**

**1- Name : احمد كمال الدين السيد عبد الرحمن  
 ID :2015170037  
 section :2**

**2- Name : احمد عصمت احمد محمد  
ID:** **2015170032  
section :2**

**3- Name : احمد مصطفى احمد المسلماني  
ID:** **2014170061  
section:3**

**4- Name :** **احمد حمدي عبد الفتاح ابراهيم   
ID:** **2014170021  
section:1**

**5- Name:احمد ناصر عبد الباقي خلاف  
ID:** **2015170054  
section:3**

**First :- Encryption / Decryption**

It’s based on two functions

1. public static RGBPixel[,] encrypt\_image(RGBPixel[,] ImageMatrix, string seed, int tap) ;

which generates 3 unique keys for each image pixel (using “GET\_Key” function) then it encrypt those pixels by Xoring pixel values with the generated keys and returns a new encrypted image.

**Function code :-**

public static RGBPixel[,] encrypt\_image(RGBPixel[,] ImageMatrix, string seed, int tap)

{

int hight = GetHeight(ImageMatrix);

int width = GetWidth(ImageMatrix);

int seed\_alpha =0;

long seed2 ;

int len;

for (int i = 0; i < seed.Length; i++)

{

if ((seed[i] != '0' && seed[i] != '1')||seed.Length>64)

{

for (int j = 0; j < seed.Length; j++)

seed\_alpha += seed[j];

seed =Convert.ToString(seed\_alpha,2);

break;

}

}

seed2 = Convert.ToInt64(seed, 2);

len = seed.Length;

byte Rkey;

byte Gkey ;

byte Bkey;

for (int i = 0; i < hight; i++)

{

for (int j = 0; j < width; j++)

{

Rkey = GET\_Key(ref seed2, tap,len);

Gkey = GET\_Key(ref seed2, tap,len);

Bkey = GET\_Key(ref seed2, tap,len);

ImageMatrix[i, j].red = (byte)(ImageMatrix[i, j].red ^ Rkey);

ImageMatrix[i, j].green = (byte)(ImageMatrix[i, j].green ^Gkey);

ImageMatrix[i, j].blue = (byte)(ImageMatrix[i, j].blue ^ Bkey);

}

}

return ImageMatrix;

}

**Function analysis :-** T(N)=N2+c = O(N2).

1. public static byte GET\_Key(ref long seed, int tap,int len)  
     
   which generates a byte key using LFSR algorithm ( by Xoring the left most bit with the bit at tap number then shift the seed left and insert the previous Xoring result , we repeat this operation 8 times to get a byte key “8 bits”) it takes the seed passed by reference & tap position and seed length then returns the generated key.

**Function Code :-**

public static byte GET\_Key(ref long seed, int tap,int len)

{

byte key = 0;

int res;

for (int i = 0; i < 8; i++)

{

res = ((((seed & (1 << len-1)) != 0)?1:0 )^ (((seed & (1 << tap)) != 0) ? 1 : 0));

key =(byte)((key<<1)+res);

seed = ((seed << 1) + res);

}

return key;

}  
**Function analysis :-** T(N)=C =O(1).

**Second :- Huffman tree construction**class Huffman Tree takes a dictionary consists of color values and it’s frequencies then store it in a priority queue and using priority queue operations it ends with one HuffmanNode containing the right and the left sub trees .  
the using function CreateEncodings() to traverse the tree and create the Huffman code for each color.  
  
**HuffmanNode class code**class HuffmanNode

{

public HuffmanNode Left { get; set; }

public HuffmanNode Right { get; set; }

public int Value { get; set; }

public int Count { get; set; }

public HuffmanNode Clone()

{

var obj = (HuffmanNode)this.MemberwiseClone();

return obj;

}

}

**Huffman Class Code :-**class HuffmanTree

{

public HuffmanNode root;

public HuffmanTree(Dictionary<int, int> counts)

{

PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<HuffmanNode>();

foreach (KeyValuePair<int, int> kvp in counts)

{

HuffmanNode N = new HuffmanNode();

N.Value = kvp.Key;

N.Count = kvp.Value;

priorityQueue.Enqueue(N, kvp.Value);

}

while (priorityQueue.Count > 1)

{

HuffmanNode n1 = priorityQueue.Dequeue();

HuffmanNode n2 = priorityQueue.Dequeue();

HuffmanNode n3 = new HuffmanNode();

n3.Left = n2;n3.Right = n1;n3.Count = n1.Count + n2.Count;

priorityQueue.Enqueue(n3, n3.Count);

}

root = priorityQueue.Dequeue();

}

public Dictionary<int, string> CreateEncodings()

{

Dictionary<int, string> Huffmancodes = new Dictionary<int, string>();

Create\_Encodings\_Help(root, "", Huffmancodes);

return Huffmancodes;

}

private void Create\_Encodings\_Help(HuffmanNode node, string code, Dictionary<int, string> Huffmancodes)

{

if (node.Left != null )

{

Create\_Encodings\_Help(node.Left, code + '0', Huffmancodes);

Create\_Encodings\_Help(node.Right, code + '1', Huffmancodes);

}

else

{

Huffmancodes.Add(node.Value, code);

}

}

}

**Class analysis :-**1- creating tree “class constructor” :-   
T(N)=2N = O(N)  
2- traversing tree “CreateEncodings()” :-  
T(N)=2T(N/2)+C ,a=2,b=2 ,f(N)=C  
∴ order = O(Nlog22) = O(N).

**Third :- Compression**compressing encrypted image in a binary file as :-  
1-writing image height , width ,seed and tap .  
2-get color frequencies for each channel then write it .  
3-get the Huffman representation for each color   
in a string list then convert it to byte list (using ConvertStringByte function)  
 and write it to file “binary stream”.

**Compression Code “two functions” :-  
1-** public static void Compress\_image(RGBPixel[,] ImageMatrix,string seed,int tap)

{

FileStream fs = new FileStream("Encrypted\_image.bin", FileMode.Create);

BinaryWriter bw = new BinaryWriter(fs);

int hight = GetHeight(ImageMatrix);

int width = GetWidth(ImageMatrix);

bw.Write(Convert.ToUInt16(hight));

bw.Write(Convert.ToUInt16(width));

bw.Write(seed);

bw.Write(Convert.ToUInt16(tap));

Dictionary<int, int> Rvalues = new Dictionary<int, int>();

Dictionary<int, int> Gvalues = new Dictionary<int, int>();

Dictionary<int, int> Bvalues = new Dictionary<int, int>();

///initiallize dictionary with 0

for (int i = 0; i < hight; i++)

{

for (int j = 0; j < width; j++)

{

Rvalues[ImageMatrix[i, j].red] = 0;

Gvalues[ImageMatrix[i, j].green] = 0;

Bvalues[ImageMatrix[i, j].blue] = 0;

}

}

for (int i = 0; i < hight; i++)

{

for (int j = 0; j < width; j++)

{

//when exist

Rvalues[ImageMatrix[i, j].red]++;

Gvalues[ImageMatrix[i, j].green]++;

Bvalues[ImageMatrix[i, j].blue]++;

}

}

bw.Write(Convert.ToUInt16(Rvalues.Count));

foreach (KeyValuePair<int,int> K in Rvalues)

{

byte[] bb = BitConverter.GetBytes(K.Value);

bw.Write(Convert.ToByte(K.Key));

bw.Write(Convert.ToByte(bb[0]));

bw.Write(Convert.ToByte(bb[1]));

bw.Write(Convert.ToByte(bb[2]));

}

bw.Write(Convert.ToUInt16(Gvalues.Count));

foreach (KeyValuePair<int, int> K in Gvalues)

{

byte[] bb = BitConverter.GetBytes(K.Value);

bw.Write(Convert.ToByte(K.Key));

bw.Write(Convert.ToByte(bb[0]));

bw.Write(Convert.ToByte(bb[1]));

bw.Write(Convert.ToByte(bb[2]));

}

bw.Write(Convert.ToUInt16(Bvalues.Count));

foreach (KeyValuePair<int, int> K in Bvalues)

{

byte[] bb = BitConverter.GetBytes(K.Value);

bw.Write(Convert.ToByte(K.Key));

bw.Write(Convert.ToByte(bb[0]));

bw.Write(Convert.ToByte(bb[1]));

bw.Write(Convert.ToByte(bb[2]));

}

HuffmanTree tree1 = new HuffmanTree(Rvalues);

HuffmanTree tree2 = new HuffmanTree(Gvalues);

HuffmanTree tree3 = new HuffmanTree(Bvalues);

Dictionary<int, string> Redtree = tree1.CreateEncodings();

Dictionary<int, string> Greentree = tree2.CreateEncodings();

Dictionary<int, string> Bluetree = tree3.CreateEncodings();

List<string> R\_binstream = new List<string>();

List<string> G\_binstream = new List<string>();

List<string> B\_binstream = new List<string>();

for (int i = 0; i < hight; i++)

for (int j = 0; j < width; j++)

{

R\_binstream.Add(Redtree[ImageMatrix[i, j].red]);

G\_binstream.Add(Greentree[ImageMatrix[i, j].green]);

B\_binstream.Add(Bluetree[ImageMatrix[i, j].blue]);

}

byte[] b = ConvertStringByte(R\_binstream).ToArray();

bw.Write(Convert.ToInt32(b.Length));

bw.Write(b);

b = ConvertStringByte(G\_binstream).ToArray();

bw.Write(Convert.ToInt32(b.Length));

bw.Write(b);

b = ConvertStringByte(B\_binstream).ToArray();

bw.Write(Convert.ToInt32(b.Length));

bw.Write(b);

bw.Close();

fs.Close();

}

**Function analysis :-**T(N)=3N2+6N+C = O(N2).

2- public static List<byte> ConvertStringByte(List<string> Slist)

{

int len = Slist.Count, index1 = 0, index2 = 0, index3 = 0;

byte elem = 0;

List<byte> bytes = new List<byte>();

while (true)

{

if (index3 == Slist[index1].Length)

{

index1++;

index3 = 0;

}

if (index2 == 8) // byte length

{

bytes.Add(elem);

elem = 0;

index2 = 0;

}

if (index1 == len) // string list length

break;

elem <<= 1;

elem += (byte)(Slist[index1][index3] - 48);

index3++;

index2++;

//bitscount++;

}

if (index2 != 0)

{

elem <<= (8 - index2);

bytes.Add(elem);

}

Slist = new List<string>();

return bytes;

}

**Function analysis:-**T(N)=N+C = O(N).

**Fourth :- Decompression**reads data from the binary file in same order of compression   
then reconstruction the Huffman tree and use the binary stream   
to traverse tree and get image color values the function takes file path and seed ,tap ,height and width passed by reference “output parameters” and returns the decompressed image .

**Function Code :-**public static RGBPixel[,] Decompress\_image(string path ,ref string seed,ref int tap,ref int height,ref int width)

{

FileStream fs = new FileStream(@path, FileMode.Open);

BinaryReader br = new BinaryReader(fs);

height = br.ReadUInt16();

width = br.ReadUInt16();

seed = br.ReadString();

tap = br.ReadUInt16();

RGBPixel[,] ImageMatrix = new RGBPixel[height, width];

Dictionary<int,int> RedTreeList = new Dictionary<int, int>();

Dictionary<int, int> GreenTreeList = new Dictionary<int, int>();

Dictionary<int, int> BlueTreeList = new Dictionary<int, int>();

List<byte> Redstream = new List<byte>();

List<byte> Greenstream = new List<byte>();

List<byte> Bluestream = new List<byte>();

uint b= br.ReadUInt16();

while(b!=0)

{

int key = br.ReadByte();

byte[] bb = new byte[4];

bb[0] = br.ReadByte();

bb[1] = br.ReadByte();

bb[2] = br.ReadByte();

bb[3] = 0;

int value = BitConverter.ToInt32(bb, 0);

RedTreeList.Add(key, value);

b--;

}

b = br.ReadUInt16();

while (b != 0)

{

int key = br.ReadByte();

byte[] bb = new byte[4];

bb[0] = br.ReadByte();

bb[1] = br.ReadByte();

bb[2] = br.ReadByte();

bb[3] = 0;

int value = BitConverter.ToInt32(bb, 0);

GreenTreeList.Add(key, value);

b--;

}

b = br.ReadUInt16();

while (b != 0)

{

int key = br.ReadByte();

byte[] bb = new byte[4];

bb[0] = br.ReadByte();

bb[1] = br.ReadByte();

bb[2] = br.ReadByte();

bb[3] = 0;

int value = BitConverter.ToInt32(bb, 0);

BlueTreeList.Add(key,value);

b--;

}

int size = br.ReadInt32();

Redstream.AddRange(br.ReadBytes(size));

size = br.ReadInt32();

Greenstream.AddRange(br.ReadBytes(size));

size = br.ReadInt32();

Bluestream.AddRange(br.ReadBytes(size));

br.Close();

fs.Close();

HuffmanTree tree1 = new HuffmanTree(RedTreeList);

HuffmanTree tree2 = new HuffmanTree(GreenTreeList);

HuffmanTree tree3 = new HuffmanTree(BlueTreeList);

HuffmanNode Redtree = tree1.root;

HuffmanNode Greentree = tree2.root;

HuffmanNode Bluetree = tree3.root;

int index = 0,count=0;

for (int i = 0; i < height; i++)

{

for (int j = 0; j < width; j++)

{

HuffmanNode N = Redtree.Clone();

while(N.Left!=null)

{

if(count==8 )

{

count = 0;

index++;

}

if((Redstream[index]&(1<<7))!=0)

N = N.Right;

else

N = N.Left;

count++;

Redstream[index] <<= 1;

}

ImageMatrix[i, j].red =(byte) N.Value;

}

}

count = 0;index = 0;

for (int i = 0; i < height; i++)

{

for (int j = 0; j < width; j++)

{

HuffmanNode N = Greentree.Clone();

while (N.Left != null)

{

if (count == 8)

{

count = 0;

index++;

}

if ((Greenstream[index] & (1 << 7)) != 0)

N = N.Right;

else

N = N.Left;

count++;

Greenstream[index] <<= 1;

}

ImageMatrix[i, j].green = (byte)N.Value;

}

}

count = 0; index = 0;

for (int i = 0; i < height; i++)

{

for (int j = 0; j < width; j++)

{

HuffmanNode N = Bluetree.Clone();

while (N.Left != null)

{

if (count == 8)

{

count = 0;

index++;

}

if ((Bluestream[index] & (1 << 7)) != 0)

N = N.Right;

else

N = N.Left;

count++;

Bluestream[index] <<= 1;

}

ImageMatrix[i, j].blue = (byte)N.Value;

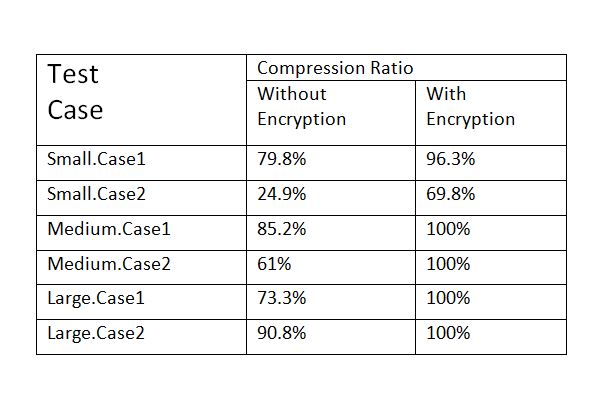
}

}

return ImageMatrix;

}

**Function analysis :-**T(N)=3N2+3N = O(N2).

****