Department of Computer Science and Engineering
Indian Institute of Technology, Indore
CS 254 - Design and Analysis of Algorithms

Encoding Algorithms and Its Application in Image Compression

Shah Miten (180001049)
Sundesh Gupta (180001057)

Contents

1. Introduction

As part of our Algorithm project, we have worked on Application of Encoding Algorithm in Image Compression. We have carefully studied and implemented Huffman coding and Arithmetic Coding. Further, We have implemented "An Efficient Encoding Algorithm Using Local Path on Huffman Encoding Algorithm for Compression." paper by Erdal et al. in 2019. All codes and results for the project can be found here. All the figures generated in this report are placed in $/figures$ directory in the github repository.

1.1 Encoding Algorithms

Encoding Algorithms are used to transform data from one form to another to achieve tasks such as compression of data. Huffman encoding and Arithmetic encoding algorithms are one of the most fundamental algorithms, and they have a wide range of applications, including image compression. Huffman Coding is used for Lossless data compression and generates prefix codes. However, in cases where a small number of input characters are used, and there is a small difference in the probability/frequency of characters, the efficiency of Huffman encoding is reduced. Arithmetic Coding addresses the deficiencies of Huffman coding, but it is challenging to implement and have slower execution. Also, no prefix codes are generated in case of Arithmetic Coding. Improved Huffman Coding that is introduced in the paper by Erdal et al. has all the benefits of Huffman Coding and achieves better compression even if probability/frequency of characters are comparable.

1.2 Image Compression

Image compression is a type of data compression that is applied to digital images to reduce their cost for storage or transmission. A set of procedures used to compress an image is called an image compression system. Compressor and decompressor operations generate an image compression system. The compressor process consists of two stages: preprocessing and encoding. Unlike the compressor process, the decompressor operation consists of the postprocessing phase after the decoding process. The encoding phase is one of the essential steps in compression algorithms. Huffman encoding and Algorithmic coding algorithms are used in the encoding phase of image compression. The main objective of compression algorithms is to reduce the number of bits needed to symbolize the characters. Image compression measurements include:

- Compression Percentage (CP):
  $CompressionPercentage = 100\left(1 - \frac{Compressedfilesize(bytes)}{Originalfilesize(bytes)}\right)$
- Number of bits per pixel (NoBPP):
  $NumberofBitsPerPixel = \frac{Numberofbits}{Numberofpixels}$

2. Huffman Coding

2.1 Algorithm Design

Huffman Coding is an entropy encoding technique. It is one of the standard algorithms. It is a frequency-based method, in which character with higher frequency gets a code of lesser length, compared to the character with lower frequency. In Huffman coding, we consider a tree-like structure. If we consider n as the number of distinct character in the text input, then we consider a binary tree with n leaf and n-1 internal node. Thus total nodes are 2*n - 1. One of the property is that every character is represented by a unique prefix code.

Ex. Message to be encode: "blvbjeanvnkjvnjlearknjnakrnekndcjbhbllivuekrjhggefskvhfbvkjdbsrjgk". Number of distinct letters in text are 17.

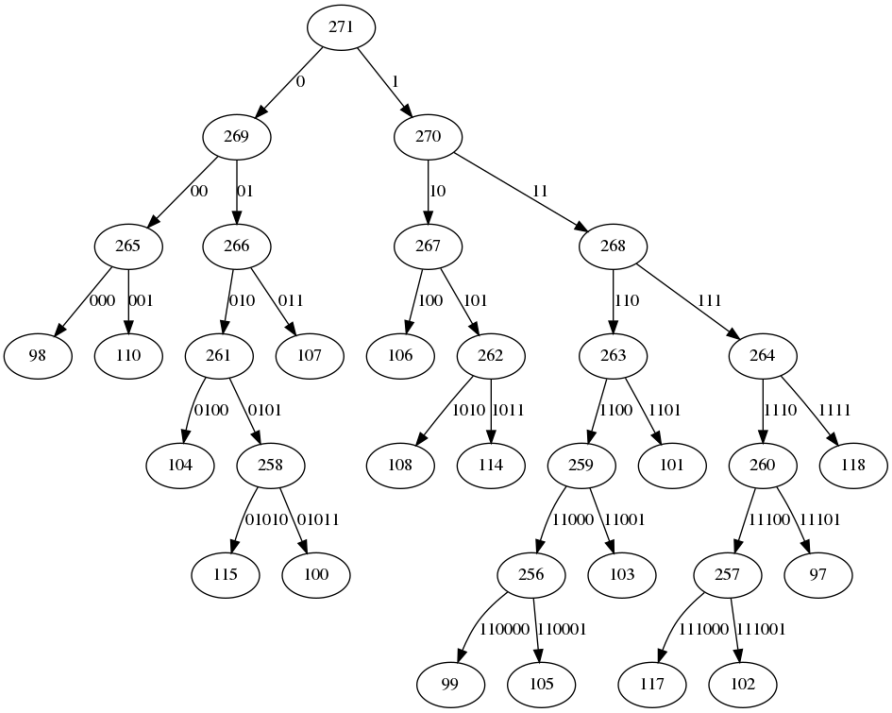


Figure 2.1: Huffman Tree

2.2 Asymptotic Analysis

Encode:

We have used "priority_queue", from standard template library of C++. The time complexity for push and pop are log(n) for both. So, the time complexity for generating tree is O(n * log(n)), where n is the number of distinct numbers in the text. Then for generating the code, we have to linearly iterate over text. Therefore the total time complexity is, O(n * log(n) + |text|). We are using a priority queue, that has a similar structure to the heap. Max numbers of elements in the priority queue are n. We are also storing the code in memory. So,
Time Complexity -> O(n * log(n) + |text|)
Space Complexity -> O(n + |code|) ≈ O(|code|)

Decode:

In decoding, First, we have to generate the Huffman tree, that is used to generate the code. For generating the tree again, it will require, approximately O(n * log(n)) (For each distinct character in the text, we have to iterate over its prefix code, at each prefix code will be a different length, but we take the sum of all of them will be approximately equal to n*log(n)). In the end, we have to linearly iterate over the code, to generate the text. We are keeping the code and tree in memory. So,

Time Complexity: O(n * log(n) + |text|)
Space Complexity: O(n + |code|) ≈ O(|code|)

3. Arithmetic Coding

3.1 Algorithm Design

Arithmetic coding is also a entropy encoding technique. Arithmetic coding is different from other forms of entropy-based encoding, like Huffman coding. In arithmetic coding rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, an arbitrary-precision fraction q where 0.0 q < 1.0.

ex. suppose if we represent A by range [0.0, 0.01), B by [0.01, 0.02), C by [0.02, 0.03), and so on. Now if we wanted to encode a string WXY, we will get the code that is between [0.222324, 0.222325).

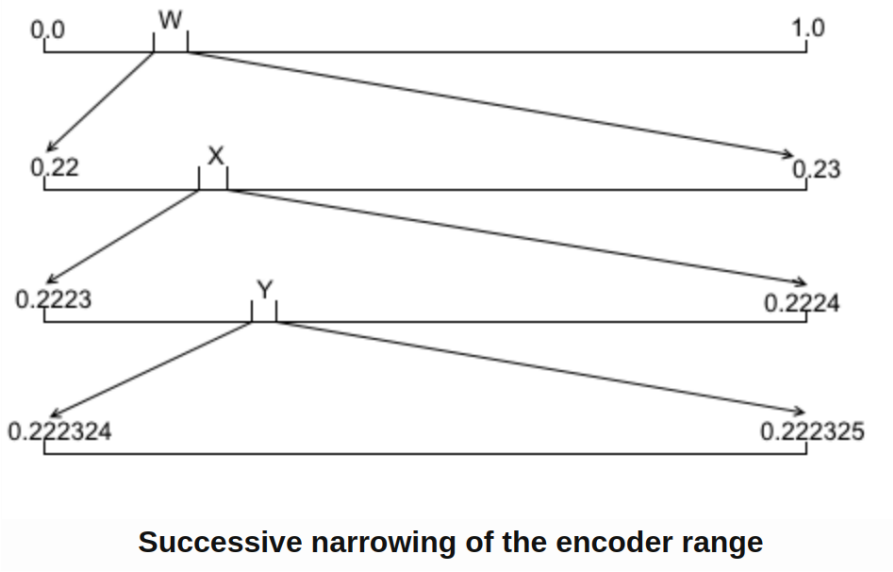


Figure 3.1: Procedure Image Source

Why is this coding optimal than naive coding?

Suppose we have a text, "ABBCAB". If we want to encode it in a normal way, then we will require 2 bit per symbol, i.e. A will be '00', B will '01', C will be '10', and '11' will be unused. So, we will require 12 bits to represent this text. Now, consider if we represent this text in base 3, then it will be represented as 0.011201_3. The next step is to encode this number using a fixed-point binary number of sufficient precision to recover it, such as 0.0010110010_2, that is 10 bits. We have saved 2 bits in comparison to the naive method. Even we optimized it further, by considering the frequency-based model, instead of equal probability.

As by this method, we get a fraction, but as any language doesn't support arbitrary-precision, we have to find an alternative method that can represent a fraction. We found a blog, https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html, that explains the method in detail.

3.2 Implementation

Even though the main idea of the algorithm is very simple, but to represent decimals as integers, we need to exploit properties of the algorithm, and also deal with the various case.

Encode:

```
string write_bits(bool bit, int bit_to_fall){
    string tmp;
  tmp += to_string(bit);
  while(bit_to_fall){
    tmp += to_string(!bit);
    bit_to_fall -= 1;
  }
  return tmp;
}

// ... in int main()

while(i < len){     // len is length of of text input.
  if(i == len-1)  current = 256;  // At the end, we are adding a buffer character
  else current = text[i];
  i += 1;

    /*
    for every i in text input,

    low[i] -> contains the lower bound of range
    high[i] -> contains the upper bound of range

    range -> [low[i], high[i])

    lower[j] -> contains the range of that jth character
    */

  ll range = high[i-1] - low[i-1] + 1;
  low[i] = low[i-1] + (range * lower[current-1]) / len;
  high[i] = low[i-1] + (range * lower[current]) / len - 1;

    /*
    const ll MIN = 0;
    const ll MAX = 65535; We are considering 16 bits in buffer, if the number of distinct
    letters increase we have to set it to 32 bits
    const ll ONE_QTR = MAX / 4 + 1;
  const ll HALF = 2 * ONE_QTR;
  const ll THREE_QTR = 3 * ONE_QTR;
    */

  while(true){
    if(high[i] < HALF){     // when high < HALF, that means high = 0...(15 bits), and we have
    flush 0 in code
      code += write_bits(0, bit_to_fall);
      bit_to_fall = 0;
    }
    else if(low[i] >= HALF){    // when low >= HALF, that means low = 1...(15 bits), and we
    have flush 1 in code
```

```
      code += write_bits(1, bit_to_fall);
      bit_to_fall = 0;
      low[i] -= HALF;
      high[i] -= HALF;
    }
    else if(low[i] >= ONE_QTR and high[i] < THREE_QTR){     // Additional case, to prevent converging of low + 1 = high, in this case loop ends in infinite loop
      bit_to_fall += 1;
      low[i] -= ONE_QTR;
      high[i] -= ONE_QTR;
    }
    else    // no digits can be further flush into memory, so break loop
      break;
    low[i] = 2 * low[i];    // we consider low = (... bits flush in memory...)000000 (base 2)
    high[i] = 2 * high[i] + 1;  // // we consider high = (... bits flush in memory...)111111 (base 2)
  }
}

ll p = low[i-1];

for(int j=15; j>=0; j--){   // Flushing the current digits of low into code
  if((p & (1LL<<j)))  code += '1';
  else code += '0';
}
```

Decode:

```

// Function to update value, the bit at count_taken of encode, needs to be added
ll add_bit(ll value, int count_taken, bool &flag, string &encode){
  bitset<16> a(value);    // bitset is part of STL of C++

  if(flag == 1){
    a.reset(0);     // set the bit at 0th position to '0'
  }
  else if(count_taken > encode.length()){
    a.set(0);   // set the bit at 0th position to '1'
    flag = 1;
  }
  else if(encode[count_taken] == '1'){
    a.set(0);
  }
  else if(encode[count_taken] == '0'){
    a.reset(0);
  }

  value = (ll)(a.to_ulong());
  return value;
}

// ... in int main()

_low[0] = MIN;
_high[0] = MAX;
ll value = to_int(0, code); // to_int adds the first 16 bits of code to value
int count_taken = 16;
bool flag = 0;

for(int i=1;i<len;i++){     // len is length of encode
  ll range = (_high[i-1] - _low[i-1] + 1);
  ll nxt = (((value - _low[i-1]) + 1) * len - 1) / range;

```

```
  int symbol;
  for(symbol = 1; lower[symbol] <= nxt; symbol++);

  _low[i] = _low[i-1] + (range * lower[symbol-1]) / len;
  _high[i] = _low[i-1] + (range * lower[symbol]) / len - 1;

  if(symbol == 256)   // if symbol is 256, that is the character we added depicting the end of
     sequence
    break;

  decode += char(symbol);

  while(true){
    if(_high[i] >= HALF){
      if(_low[i] >= HALF){
        value -= HALF;
        _low[i] -= HALF;
        _high[i] -= HALF;
      }
      else if (_low[i] >= ONE_QTR && _high[i] < THREE_QTR)
      {
        value -= ONE_QTR;
        _low[i] -= ONE_QTR;
        _high[i] -= ONE_QTR;
      }
      else
      {
        break;
      }
    }
    _low[i] = 2 * _low[i];
    _high[i] = 2 * _high[i] + 1;
    value = add_bit(2 * value, count_taken, flag, code);    // Adding the bit at count taken
    count_taken++;
  }
}
```

3.3 Asymptotic Analysis

Encode:

The outer while loop runs in linear time, to the length of the input text. The inner while loop runs linear to the number of bits added for the ith character in the text. So overall, the time complexity of the algorithm is linear to the length of code. But as there are many mathematical calculations involved, the time constant is very high.

Time complexity -> O(|code|)
Space complexity -> O(|code|)

We can further prove the complexity, by this graph. The graph is almost linear. If we consider, O(n) $= c * n$, then the time constant for this code is approximately $3.41 * 10^{-7}$. Number of operation on an average computer is around $10^8 - 10^9$. Even if we consider 10^8 operations per second, we can see that constant ≈ 34.1, which is very high.

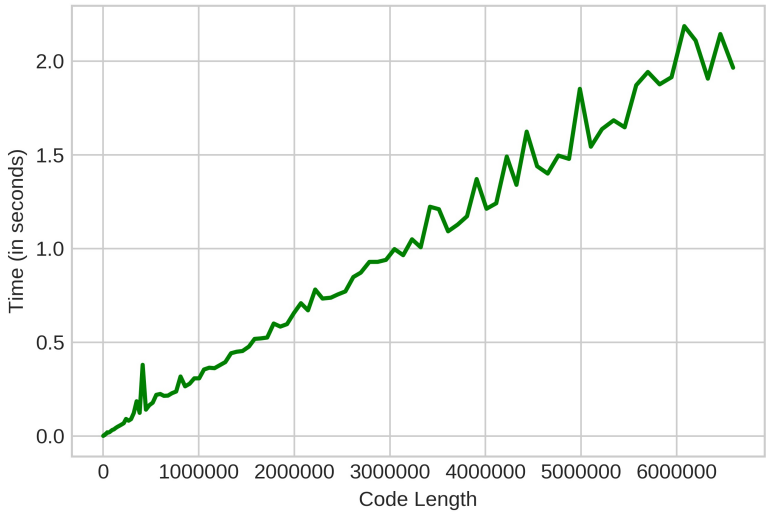


Figure 3.2: Time Complexity of Arithmetic encoding algorithm

Decode:

The outer for loop in decoding algorithm runs in linear time to the length of the input text. Inner for loop, that involves symbol runs at the worst-case complexity in the order of the number of distinct words in the text. The inner while loop runs for a total of length of code. Even this involves many mathematical operations, so the time constant for the algorithm is very high. So,

Time complexity: O(|text| * n + |code|) ≈ O(|text| * n)
Space complexity: O(|code|)

4. Improved Huffman Coding

4.1 Algorithm Design

Initially, we apply the Huffman encoding algorithm to the input text. The characters represented by the short bit sequence in the Huffman encoding algorithm are neglected. Flag bits are then added according to whether the succeeding symbols are on the same leaf. If the subsequent character is not on the same leaf, flag bit "0" is appended. Otherwise flag bit "1" is inserted between the characters.

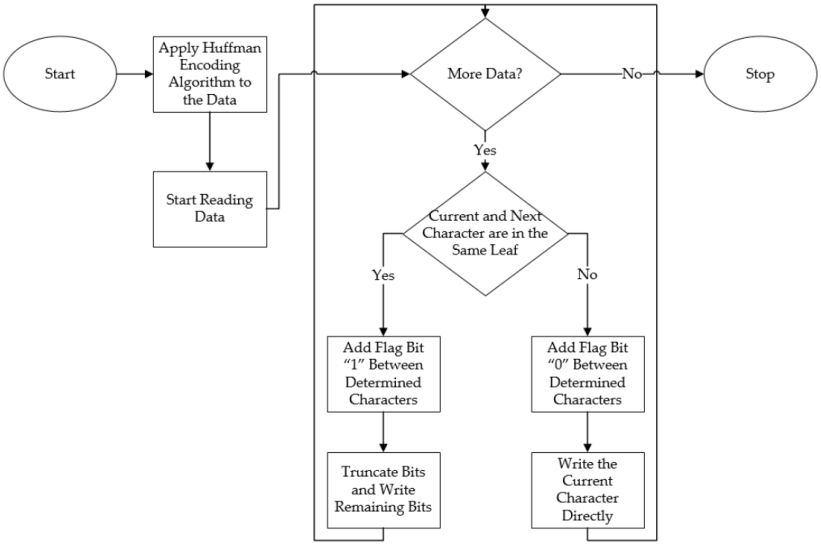


Figure 4.1: Flowchart of Improved Encoding Algorithm Image Source

Ex. consider text = "aaaabbbbcdefghjklmnoprsaabb", then the huffman tree generated by code is given in figure 4.2. The huffman encoding generated by the text will contain 98 bit, and the code will be: "00000000010101011001010011101001010110110101111100011001110101101111100111011111011111100000000101". Now, consider figure 4.3, We consider a code small if its length is smaller than 3. So in our example, a and b have small codes. So we won't use any flag bit after a. Now, we can see that c and d, are on the same leaf. In this algorithm, we define nodes on the same leaf as if they are at level 3 are same(No particular definition is given in the paper). So, we add a flag bit 1, and we truncate first 3 bit of d. Similarly, for others. The resulting bit sequence contains 83 bits, and is as follows: "00000000010101011001011101010010111011101100010111011101110010111011101010000000000101".

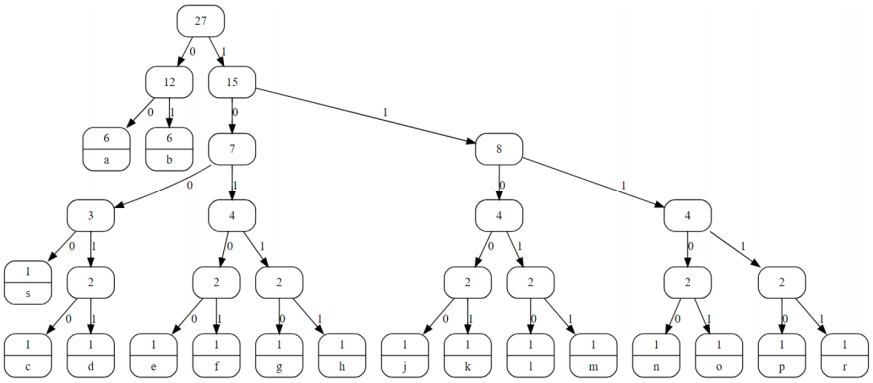


Figure 4.2: Huffman Tree Image Source

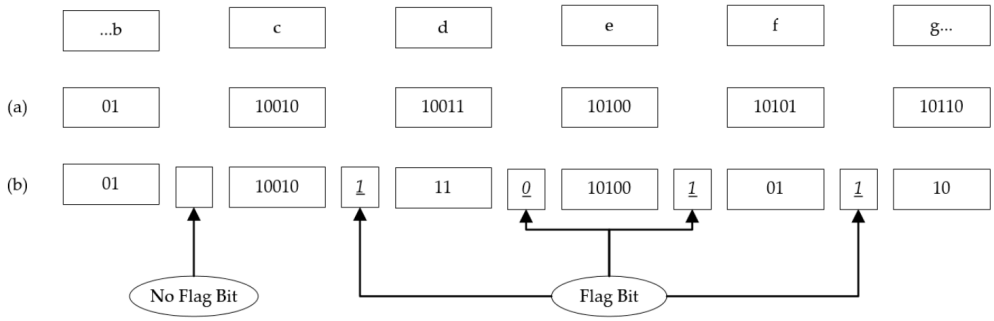


Figure 4.3: Bit sequence of encoding algorithm (a) Huffman encoding algorithm (b) Improved Huffman encoding algorithm Image Source

4.2 Implementation

Encode:

The following steps are followed to obtain the proposed algorithm.

1. Huffman encoding algorithm is applied to data.
2. Characters identified by small bits are ignored.
3. Charaters identified by large bits are followed by the flag bit. If the current and next character, have same parent at level 3, then flag bit is 1, else 0.
4. When the flag bit is 1, there is no need to repeat the three bits common in these character.

So the encoding algorithm looks like this,

```
// After Generating Huffman Tree

/*
struct huffcode{
  int label;
  int parent;
  string code;
} codes[MAX_ASCII];
*/

string code;
int last = -1;
bool f = 0;

for(int i = 0; i < s.length(); i++){
  if(f == 0){          //Case where previous char was less than 3 nodes deep in Huffman Tree
    code += codes[s[i]].code;
  }
  else if(codes[s[i]].parent == codes[last].parent){ // Same Leaf
    code += '1';   // Flag bit '1' added
    string tmp = codes[s[i]].code;
    code += tmp.substr(3, code.length()); // add code without the prefix
  }
  else{              // Not in same leaf
    code += '0';   // Add Flag bit '0'
    code += codes[s[i]].code;  // Add complete code
  }
  if(codes[s[i]].parent == -1)
    f = 0;
  else{       // if curr char is more than 3 nodes deep, set f = 1
    last = s[i];
    f = 1;
  }
}
```

Decode:

After applying Huffman Decoding Algorithm, we will do some case handling for additional flag bit.

```
pixel* ptr = root;   // Huffman Tree root
pixel* tmp = root;   // Temporary pointer

vector<int> decode;
bool f = 0;  // to check whether current bit in encoded string is Flag bit or Non-Flag bit
int level = 0;

for(auto i:w){      // for each bit in encoded string

  if(f){          // if previous char in decoded string was atleast 3 nodes deep, next bit in
    encoded string would be Flag bit
    if(i == '0'){   // Flag bit is '0'
      // do nothing
    }
    else{       // Flag bit is '1'
      ptr = tmp;  // make jump from current node to node where common prefix of curr char and
    prev char ended
      level = 3;
    }
    f = 0;    // next bit would be Non Flag
    continue;
  }

```

```
  if(i=='0')    // go to left in Huffman Tree if current bit is '0' else go to right
    ptr = ptr->left;
  else
    ptr = ptr->right;

  level += 1;

  if(level == 3)  // if 3 nodes deep in huffman tree, set temporary pointer
    tmp = ptr;

  if(ptr->label < 256){ // ptr reaches leaf node in huffman tree

    decode.push_back(ptr->label); // push decoded char

    // Reset Variables
    level = 0;
    if(flag[ptr->label])        // Next bit would be Flag bit
      f = 1;
    ptr = root;
  }
}
```

4.3 Asymptotic Analysis

Encode

The main advantage of this algorithm over Huffman encoding is, it increases the compression ratio, without any increase in complexity. We just add some extra conditions. So,

Time Complexity -> O(n * log(n) + |text|)
Space Complexity -> O(|code|)

Decode:

Similarly, decoding is achieved simply by adding some extra conditions, so the complexity is the same as the decoding complexity of the Huffman coding algorithm.

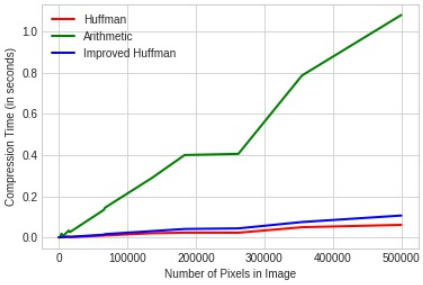
Time Complexity: O(n * log(n) + |text|)
Space Complexity: O(|code|)

5. Results

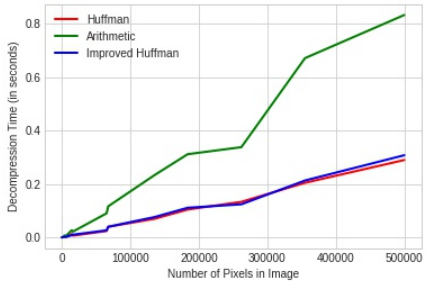
5.1 Compression and Decompression Time

Table 5.1: Grayscale Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CT (in s)	DT (in s)	CT (in s)	DT (in s)	CT (in s)	DT (in s)
1	256	0.0004	0.0009	0.019	0.0008	0.0174	0.0006
2	1024	0.0009	0.001	0.0116	0.0014	0.0181	0.0015
3	2304	0.0013	0.0014	0.0035	0.003	0.0011	0.0015
4	4096	0.0017	0.003	0.4371	0.0078	0.0131	0.0028
5	6400	0.0017	0.0026	0.0292	0.0061	0.014	0.0041
6	14672	0.0026	0.0102	0.2482	0.0279	0.0209	0.0104
7	16384	0.0015	0.008	0.1399	0.0213	0.0204	0.0084
8	65536	0.0078	0.0255	0.2272	0.0901	0.0141	0.0261
9	67859	0.0107	0.0396	0.1765	0.117	0.0242	0.0408
10	135762	0.0176	0.0877	0.3231	0.2558	0.0826	0.0757
11	183768	0.0265	0.128	0.4085	0.3134	0.0546	0.109
12	262144	0.0267	0.139	0.433	0.3514	0.0649	0.1209
13	355008	0.0536	0.2542	0.8466	0.6849	0.0962	0.2123
14	499840	0.0763	0.3502	1.1505	0.8341	0.1114	0.2925



(a) Compression Time vs Number of Pixels

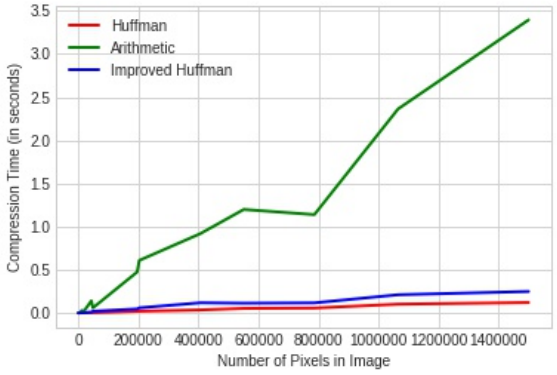


(b) Decompression Time vs Number of Pixels

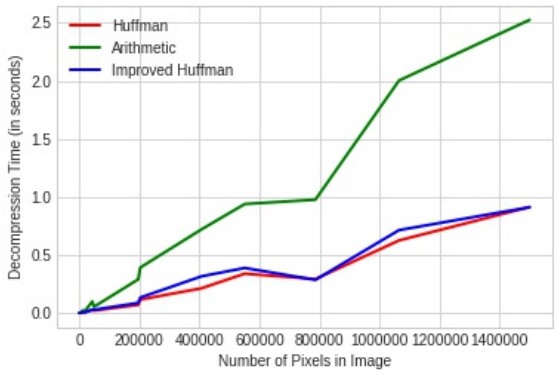
Figure 5.1: Graphs For Table 5.1

Table 5.2: RGB Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CT (in s)	DT (in s)	CT (in s)	DT (in s)	CT (in s)	DT (in s)
1	768	0.0006	0.001	0.0015	0.0022	0.0006	0.001
2	3072	0.001	0.003	0.0038	0.0036	0.0017	0.0018
3	6912	0.0025	0.0031	0.0088	0.0074	0.0017	0.0031
4	12288	0.0019	0.0071	0.029	0.0229	0.0031	0.0076
5	19200	0.0038	0.0082	0.0213	0.018	0.0033	0.0068
6	44016	0.0075	0.0268	0.1408	0.1004	0.0084	0.0292
7	49152	0.0034	0.0216	0.0594	0.0536	0.02	0.0284
8	196608	0.0196	0.0702	0.4753	0.2905	0.0472	0.0874
9	203577	0.0206	0.1162	0.609	0.3922	0.0598	0.1337
10	407286	0.0356	0.2136	0.9198	0.721	0.1185	0.3166
11	551304	0.0535	0.3395	1.1989	0.9394	0.1139	0.3886
12	786432	0.0573	0.2931	1.1393	0.9766	0.1175	0.2858
13	1065024	0.1024	0.6258	2.3624	2.0032	0.2115	0.7147
14	1499520	0.1215	0.9133	3.3936	2.5252	0.2495	0.9115



(a) Compression Time vs Number of Pixels



(b) Decompression Time vs Number of Pixels

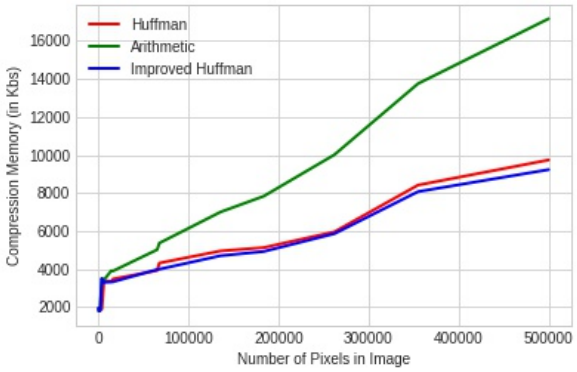
Figure 5.2: Graphs For Table 5.2

- As we have discussed in section 2.2 and 4.3, the time complexity of the Huffman and Improved Huffman are equal.
- Also we discussed in section 3.3 about large time constant of Arithmetic algorithm. One of the advantage of Huffman Algorithm over Arithmetic Algorithm is computational complexities.
- Also we can see the time constant for Improved Huffman is greater than Huffman algorithm. That is because of the extra case handling, we had to do because of flag bit.

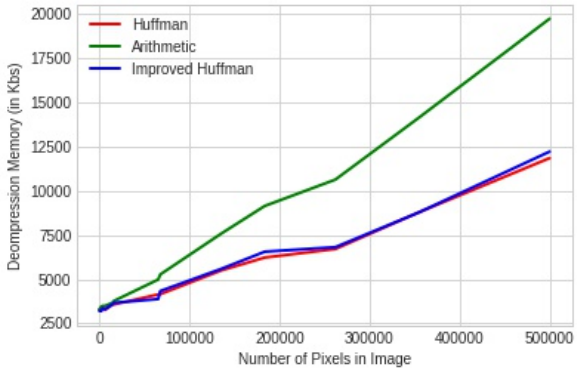
5.2 Compression and Decompression Memory

Table 5.3: Grayscale Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CM (Kbs)	DM (Kbs)	CM (Kbs)	DM (Kbs)	CM (Kbs)	DM (Kbs)
1	256	1940	3256	1808	3420	1940	3248
2	1024	1896	3160	1808	3344	1792	3308
3	2304	1896	3268	1868	3264	1816	3184
4	4096	1896	3472	3516	3500	3508	3372
5	6400	3392	3388	3428	3448	3304	3348
6	14672	3352	3588	3924	3792	3324	3560
7	16384	3480	3508	3900	3852	3332	3752
8	65536	3928	4004	5024	4876	3980	3736
9	67859	4320	4024	5368	5188	3996	4396
10	135762	4956	5468	7004	7588	4704	5468
11	183768	5132	6304	7836	9100	4928	6560
12	262144	5960	6544	10016	10760	5864	6624
13	355008	8420	8516	13756	14220	8076	8868
14	499840	9740	11760	17168	19688	9232	12232



(a) Compression Memory vs Number of Pixels

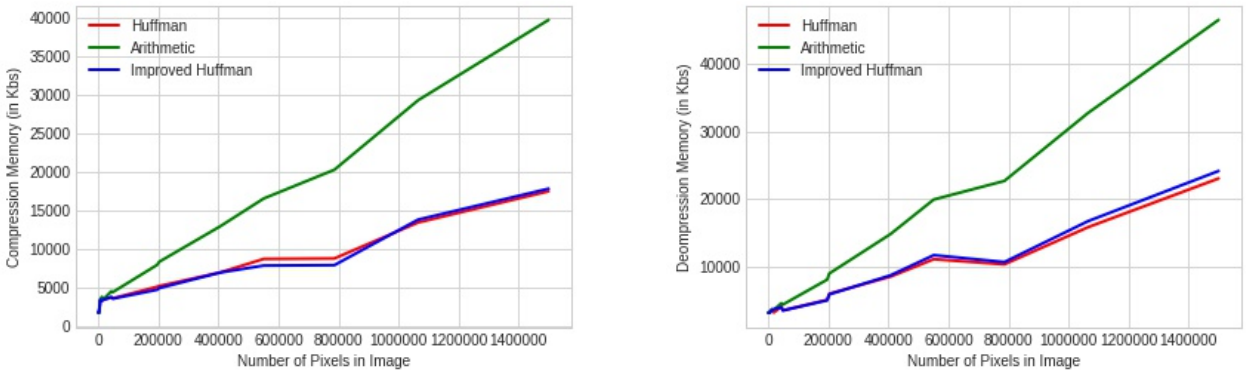


(b) Decompression Memory vs Number of Pixels

Figure 5.3: Graphs For Table 5.3

Table 5.4: RGB Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CM (Kbs)	DM (Kbs)	CM (Kbs)	DM (Kbs)	CM (Kbs)	DM (Kbs)
1	768	1792	3320	1824	3404	1932	3264
2	3072	1896	3300	1820	3316	1828	3356
3	6912	3428	3336	3456	3500	3496	3364
4	12288	3396	3692	3884	3808	3392	3804
5	19200	3504	3340	3624	3756	3520	3240
6	44016	3820	4088	4600	4720	3852	4004
7	49152	3692	3684	4496	4524	3640	3752
8	196608	5188	5248	8016	8196	4800	5100
9	203577	5280	5988	8416	9092	5008	5892
10	407286	7048	8504	13020	14888	7028	8768
11	551304	8780	11232	16600	19972	7924	11676
12	786432	8840	10440	20316	22704	7976	10660
13	1065024	13492	15876	29300	32704	13860	16720
14	1499520	17532	22940	39700	46392	17864	24320



(a) Compression Memory vs Number of Pixels

(b) Decompression Memory vs Number of Pixels

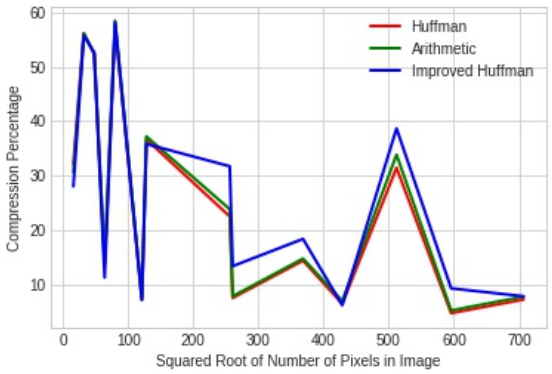
Figure 5.4: Graphs For Table 5.4

- Memory used by the program can be calculated by reading the pseudo-filesystem "/proc" and kernel calls. In Linux, the filesystem file at "/proc/self/status" contains the metadata for the current running process. This file can be accessed using "sys/sysinfo.h" library for C++.
- We can see that space complexity linearly increases as the number of pixels increase. This is because, all the space complexities are O(|code|), and length of code directly depends on the number of pixels.
- As we discussed in section 2.2 and 4.3, both have same space complexity.
- The memory constant of Arithmetic Coding is greater than Huffman and Improved Huffman method.

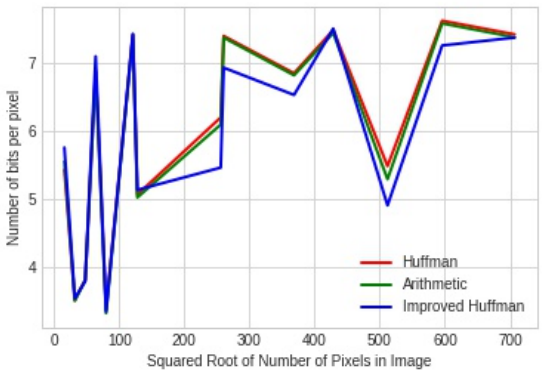
5.3 Compression Ratio and Number of bits per pixel

Table 5.5: Grayscale Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CP	NoBPP	CP	NoBPP	CP	NoBPP
1	256	32.13	5.43	30.62	5.55	28.03	5.76
2	1024	56.2	3.5	56.24	3.5	55.81	3.54
3	2304	52.37	3.81	52.46	3.8	52.66	3.79
4	4096	13.86	6.89	14.17	6.87	11.23	7.1
5	6400	57.8	3.38	58.54	3.32	58.22	3.34
6	14672	7.21	7.42	7.45	7.4	7.08	7.43
7	16384	36.69	5.07	37.22	5.02	35.83	5.13
8	65536	22.48	6.2	23.8	6.1	31.72	5.46
9	67859	7.46	7.4	7.77	7.38	13.31	6.93
10	135762	14.33	6.85	14.71	6.82	18.33	6.53
11	183768	6.39	7.49	6.93	7.45	6.12	7.51
12	262144	31.44	5.48	33.85	5.29	38.68	4.91
13	355008	4.66	7.63	5.15	7.59	9.22	7.26
14	499840	7.17	7.43	7.7	7.38	7.77	7.38



(a) Compression Percentage vs Square Root Number of Pixels

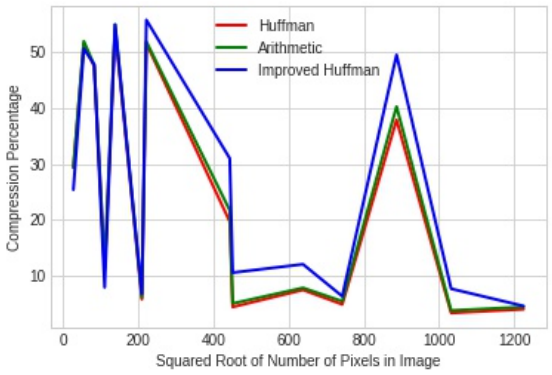


(b) No. of Bits Per Pixel vs Square Root Number of Pixels

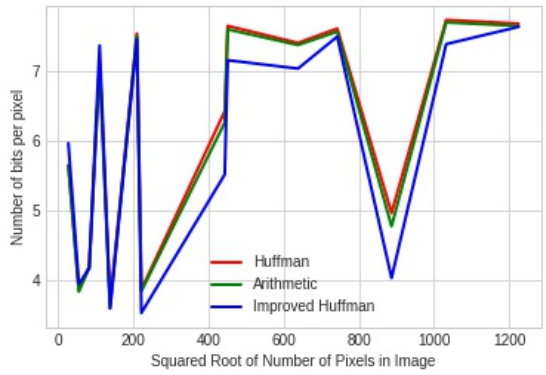
Figure 5.5: Graphs For Table 5.5

Table 5.6: RGB Images

		Huffman		Arithmetic		Improved Huffman	
S.No	No. of Pixels in Image	CP	NoBPP	CP	NoBPP	CP	NoBPP
1	768	29.39	5.65	29.36	5.65	25.41	5.97
2	3072	51.59	3.87	51.95	3.84	50.61	3.95
3	6912	47.4	4.21	47.68	4.19	47.76	4.18
4	12288	11.22	7.1	11.55	7.08	7.95	7.36
5	19200	54.09	3.67	54.93	3.61	54.9	3.61
6	44016	5.82	7.53	6.17	7.51	6.81	7.46
7	49152	51.56	3.88	51.87	3.85	55.76	3.54
8	196608	19.71	6.42	21.73	6.26	31.0	5.52
9	203577	4.44	7.64	5.07	7.59	10.58	7.15
10	407286	7.49	7.4	7.86	7.37	12.08	7.03
11	551304	4.91	7.61	5.47	7.56	6.36	7.49
12	786432	37.89	4.97	40.26	4.78	49.52	4.04
13	1065024	3.37	7.73	3.82	7.69	7.7	7.38
14	1499520	4.04	7.68	4.44	7.64	4.62	7.63



(a) Compression Percentage vs Square Root Number of Pixels



(b) No. of Bits Per Pixel vs Square Root Number of Pixels

Figure 5.6: Graphs For Table 5.5

- We can see that Improved Huffman Coding performed well compared to Huffman and Arithmetic Coding. In the paper, they used some pre-processing on Images and didn't share any information about that in the paper, so their result is marginally better than ours.
- In general Arithmetic Coding performs better than Huffman Coding. This is one of the advantages of Arithmetic Coding over Huffman Coding.
- When numbers of pixels in images are less, all the algorithms produce almost the same results.

6. Conclusion

In this project, we carefully studied Huffman Coding, Arithmetic Coding and Improved Huffman Coding. We observed that Improved Huffman Encoding performs better than both Huffman and Arithmetic Coding. We also saw, even though Arithmetic Coding performs better than Huffman Coding, it is computationally very expensive. Also, A variety of specific techniques for arithmetic coding have historically been covered by US patents. Further, Huffman Coding is a freely available Code, and so many variants of it are practically used. It is used in DEFLATE (a data compression file format) and PNG (file format for images). Hence, incorporating the Improved Huffman algorithm in these widely known techniques can be valuable.