*Project Report*

On

Data compression

Submitted by

**Mahesh Kumar 160001033**

**K. Nitin Kumar 160001027**

Computer Science and Engineering

2nd year

Under the Guidance of

**Dr. Kapil Ahuja**



Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2017

**Index**

**1.** Run lenght encoding

1. Introduction
2. Working mechanism
3. Real world useWo
4. Algorithm Analysis
5. Pseudo code
6. Actual Code
7. Complexity analysis
8. Time complexity
9. Space complexity

**2. Huffman coding**

* + - 1. Introduction
      2. Working mechanism
      3. Real world use
      4. Algorithm analysis
      5. Pseudo code
      6. Actual code
      7. Complexity analysis

1. Space complexity
2. Time complexity

**3. LZW compression**

1. Introduction
2. Working mechanism
3. Real world use
4. Algorithm analysis
5. Pseudo code
6. Complexity analysis
7. Time complexity
8. Space complexity
9. Optimisations
10. Implementations and results

***4. Future Works and scopes***

**Algoriths:-**

**1.Run length encoding (R.L.E compression)**

RLE stands for Run Length Encoding. It is a lossless algorithm that only offers decent compression ratios in specific types of data.

How RLE works

RLE is the easiest compression algorithm. In RLE data is compressed by eliminating redundant charcters or elements.

For example:- let we have given ADBBBBBBCDDDDDEEEE

We can write it in RLE as

AD \*6BC \*5D \*4E

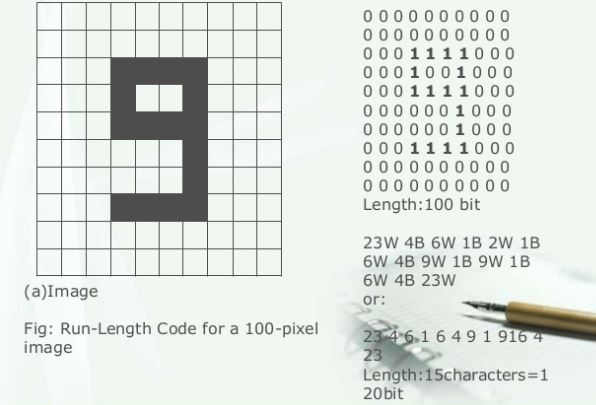
Note: We can use any character in place of \* which is not present in the string.

Efficiency depends on number of repeated charcters in the data.

It is very useful where there is repetitive character occurs like in images where same type of colors appears in the areas. It is used as an element in more complex image compression technique.

For run length encoding on an image, transmission of digital line scan is replaced by transmission of a quantity count of each of successive run of black and white scan picture element.

Here is the basic example of image compression using RLE



* **Algorithm analysis:-**

Pseudo code-

Find frequency of each character which is consecutively repeated.

If (repetition<=2)

Leave as it is.

Else

{

Put a symbol

Print that character with its frequency

}

Actual code-

#include<bits/stdc++.h>

#include<iostream>

#include<string>

#include<vector>

using namespace std;

int main()

{

string s, ans;

cout<<"Enter the data: ";

cin>>s;

char t=s[0];

int count=0;

vector<char> v;

vector<int> v1;

v.push\_back(t);

for(int i=0; i<s.size(); i++)

{

if(t == s[i])

{

count++;

}

else

{

if(count>2)

{

v.pop\_back();

v.push\_back(' ');

v.push\_back('\*');

v1.push\_back(count);

v.push\_back(t);

v.push\_back(s[i]);

}

else if(count==2)

{

v.push\_back(t);

}

else

{

v.push\_back(s[i]);

}

t=s[i];

count=1;

}

}

if(count==2)

{

//v.push\_back(t);

v.push\_back(t);

}

else if(count>2)

{

v1.push\_back(count);

v.pop\_back();

v.push\_back(' ');

v.push\_back('\*');

v.push\_back(t);

}

int j=0;

for(int i=0; i<v.size(); i++)

{

if(i>=1 && v[i-1]=='\*')

{

cout<<v1[j];

j++;

}

cout<<v[i];

}

return 0;

}

Time complexity:- O(n)

Space complexity:- O(n)

2.Huffman coding

Huffman coding is a lossless data compression greedy algorithm. It was created by an MIT student named David Huffman in 1952.

How Huffman coding works

It use the probability distribution of the alphabet of the source to develop code words for symbols. First we calculate the frequency distribution of all the characters of the source or text. According to frequency code word is assigned. Shorter code words for higher frquencies and longer code words for smaller frequencies are assigned. For this work a binary tree is created using the symbols as leaf nodes according to their frequency and edges are taken as code word.

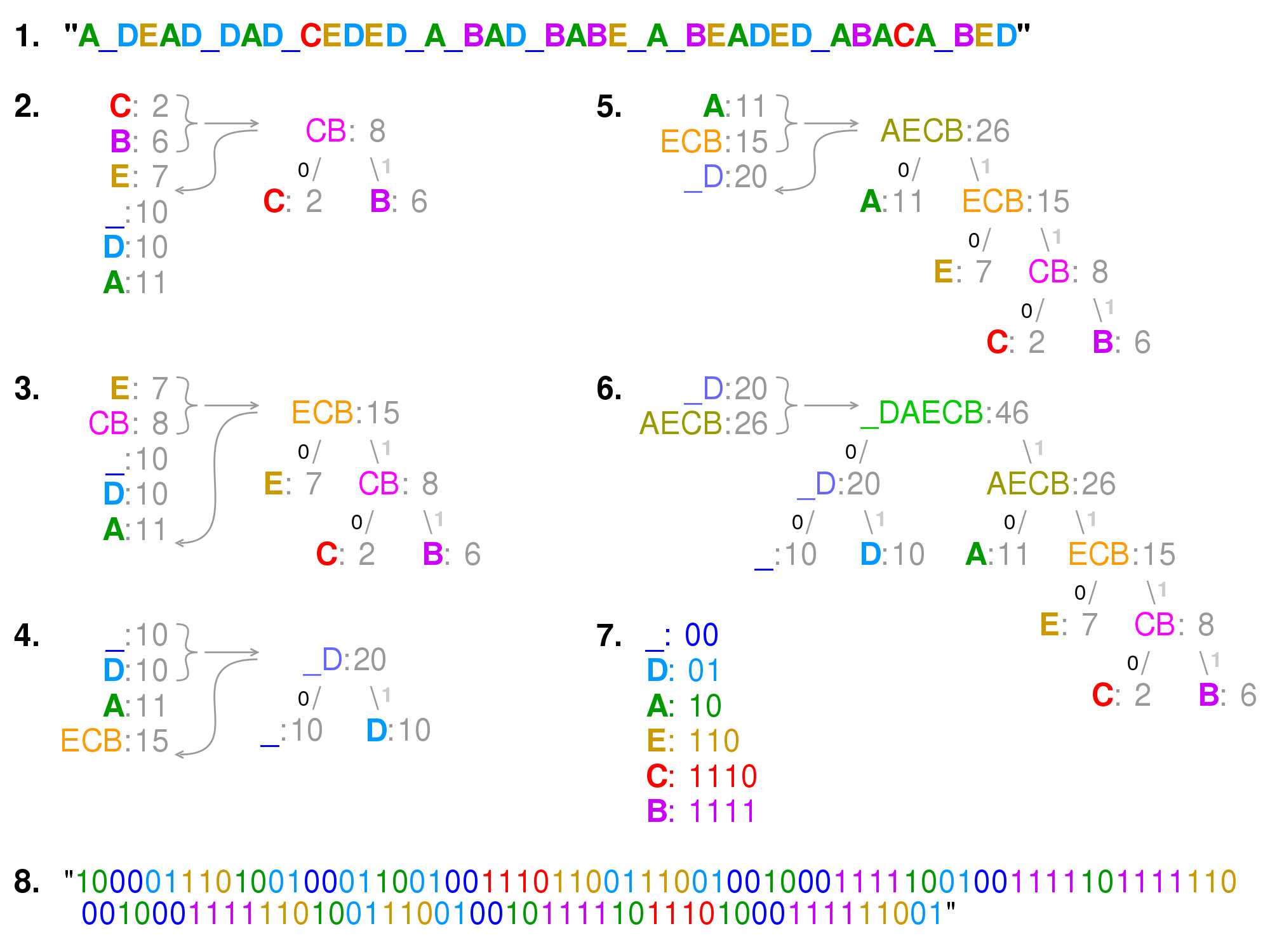
For example let a string A\_DEAD\_DAD\_CEDED\_A\_BABE\_A\_BEADED\_ABACA\_BED

No. of characters= 42

Space used= 42 bytes= 42\*4 = 168 bits.

Real world use:-

**ZIP** is the most widely used compression tool that uses Huffman Encoding as its basis. The latest of the most efficient lossless compression algorithms, Brotli Compression, released by Google last month also uses Huffman coding.



Before Huffman coding space=46 bytes=46\*4 = 184 bits.

After Huffman coding,

Space = 117 bits.

* Algorith analysis

Pseudo code:

-First iterate the string.

-Find the frequencies of each element in the whole string.

-Sort(elements according to frequency)

-Put lowest and 2nd lowes to the leaf node

Let say f1 and f2

-The parent of these two upper child will have frequency f1+f2.

-Iterate till we reach the final parent

-Now assign left edge to ‘0’ and right edge to ‘1’.

Now each element is assigned a unique code by traversing from parent to that child.

Actual code:-

#include <bits/stdc++.h>

using namespace std;

struct MinHeapNode

{

char data;

unsigned freq;

MinHeapNode \*left, \*right;

MinHeapNode(char data, unsigned freq)

{

left = right = NULL;

this->data = data;

this->freq = freq;

}

};

struct compare {

bool operator()(MinHeapNode\* l, MinHeapNode\* r)

{

return (l->freq > r->freq);

}

};

void printCodes(struct MinHeapNode\* root, string str)

{

if (!root)

return;

if (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

void HuffmanCodes(char data[], int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare> minHeap;

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

minHeap.push(new MinHeapNode(data[i], freq[i]));

while (minHeap.size() != 1)

{

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = new MinHeapNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

printCodes(minHeap.top(), "");

}

int main()

{

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 12, 13, 16, 45 };

int size = sizeof(arr) / sizeof(arr[0]);

HuffmanCodes(arr, freq, size);

return 0;

}

Complexity:-

The time complexity of the Huffman algorithm is **O(nlogn)**.

Using a heap to store the weight of each tree, each iteration requires **O(logn)** time to determine the cheapest weight and insert the new weight. There are **O(n)** iterations, one for each item.

space complexity:- **O(n)**

3.)LZW compression:-

**Algorithm name: Lempel – Ziv – Welch (LZW)**

Lempel–Ziv–Welch (**LZW**) is a universal lossless data**compression** algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch in 1984 . The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and optionally in PDF and TIFF. Unix’s ‘compress’ command, among other uses. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format.

**Speed and compression:-**

The speed of compression as well as the size of compressed file depends a lot on input file and the dictionary built up, stored and the way it is accessed. As the number of keys in the dictionary increases, a good idea would be to use hash tables to speed up access and storage. At the same time, when the number of keys increase decompression takes more time to access the values.

**Pseudocode:**

**Compression:**

Initialize the dictionary (with the first 256 entries).

[prefix\_string] ← [empty]

While (bytes left in input) {

B ← next byte in the input.

Is the string [prefix\_string]B in the dictionary?

Yes:

[prefix\_string] ← [prefix\_string]B

No:

Add the string [prefix\_string]B to the dictionary.

Output the index of [prefix\_string] to the compressed file.

[prefix\_string] ← B

}

Output the index of [prefix\_string] to the compressed file.

**Decompression:**

Initialize the dictionary ‘string[]’ (with the first 256 entries).

<index> ← first index value in the input.

Write string[index] to the result.

[old] ← string[index]

While (bytes left in input) {

<index> ← next index value in the input.

Does <index> exist in the dictionary?

Yes:

Write string[index] to the result.

B ← string[index][0]

Add [old]B to the dictionary.

No:

B ← old[0]

Add [old]B to the dictionary.

Write the string for [old]B to the decompressed file.

[old] ← string[index]

}

**Complexity Analysis:**

A simple implementation on the above algorithm takes O(n2) time for compression and O(n) time for decompression.

**Justification –**

1. In the compression algorithm, the outer while loop –

**“While (bytes left in input) “**

takes O(length of file) time to run. Inside this while loop, searching a string in the dictionary –

**“Is the string [prefix\_string]B in the dictionary?”**

Takes at max O(length of file) time to run as in the worst case, the dictionary can fill like this –

We know that 256 characters are already stored in the dictionary.

For the first 256 x 256 characters, the dictionary may fill at every new character, as each sequence of 2 characters may not be present.

For the next 256 x 256 x 256 characters, the dictionary will fill after every 2 characters, as every possible sequence of 2 characters is present in the dictionary.

So, the size of dictionary (in the worst case), varies with ‘n = length of file’ as –

This simply turns out to be O(n), as for any n, the formula simply computes to a form where n. So, searching for a string in dictionary turns out to be of linear time complexity.

Hence, the net result is O(n2) running time.

1. In the decompression algorithm, there is only an outer while loop –

**“While (bytes left in input)”**

that takes O(length of compressed file) time to run. Inside the while loop, searching an index in the dictionary –

**“Does <index> exist in the dictionary?”**

is a simple O(1) operation as it just checks whether the array size is atleast ‘**<index>**’ length long.

**Optimisations:-**

The naive implementation of the LZW algorithm takes O(n2) time to compress a given text file, as discussed above. However, it can be optimized and can be done in linear time, using trie data structure.

Every node of the trie will be a structure containing 3 fields –

|  |
| --- |
| **char** symbol |
| **int** dict\_index |
| **node** \*children[256] |

**Pseudocode –**

Initialize the dictionary ‘root’

for (i = 0 to 255) {

root[i] -> symbol = character with ascii value ‘i’.

root[i] -> dict\_index = i;

initialise all children pointers to null;

}

B = first character in file;

node \*currentnode = root[ascii(B)];

int nextindex = 256;

While (bytes left in input) {

B ← next byte in the input.

Is a node with symbol = B present as a child of currentnode?

Yes:

currentnode = child node of currentnode with symbol = B;

No:

Create a new node ‘temp’ with symbol = B, dict\_index = nextindex, and all children pointers NULL;

nextindex++;

currentnode->children[ascii(B)] = temp;

Output the dict\_index of currentnode to the compressed file;

currentnode = root[ascii(B)];

}

Output the dict\_index of currentnode to the compressed file;

**Complexity Analysis –**

We see that now, with every new character read from the uncompressed file, we only check the next character, whether it is present in the trie, as a child of currentnode, which keeps updating itself with every passing letter. So, we do not need to check the whole string again and again.

The above implementation takes O(1) time to search and create the dictionary, along with an outer while loop that iterates through the length of the file.

Hence the optimised overall time complexity for the compression algorithm now becomes O(n).

**Implementation and Results**

The naïve LZW algorithm was implemented in python, whereas the efficient LZW algorithm was implemented in C++. Text files with varying sizes were compressed using the algorithm.

Data compression ratio =

The results of the naïve algorithm were as follows-

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of file | Original Size | Time taken to compress | Compressed file size | Compression Ratio | Time taken to decompress |
| small.txt | 14.8 KB | 1.07 seconds | 8.42 KB | 1.75 | 0.015 seconds |
| medium.txt | 93.1 KB | 16.25 seconds | 33.5 KB | 2.78 | 0.046 seconds |
| large.txt | 162 KB | 41.29 seconds | 59.2 KB | 2.73 | 0.062 seconds |

The results of the efficient algorithm were as follows-

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of file | Original Size | Time taken to compress | Compressed file size | Compression Ratio | Time taken to decompress |
| small.txt | 14.8 KB | 0.056 seconds | 8.42 KB | 1.75 | 0.015 seconds |
| medium.txt | 93.1 KB | 0.074 seconds | 33.5 KB | 2.78 | 0.046 seconds |
| large.txt | 162 KB | 0.107 seconds | 59.2 KB | 2.73 | 0.062 seconds |

As a result, we can see a significant improvement in compression times, by using an efficient implementation of the LZW algorithm.

**Future Work:-**

Compression algorithms can be further improved by using various special techniques.

DEFLATE is a modern data compression algorithm, used in the .zip file format. It improves LZSS by introducing Huffman codes. It is also based on the standard bit manipulation to denote the encoded and not coded streams.

Compression is achieved through two steps:

1. The matching and replacement of duplicate strings with pointers.
2. Replacing symbols with new, weighted symbols based on frequency of use.

Machine learning can also help in compressing data. A system that predicts the posterior probabilities of a sequence given its entire history can be used for optimal data compression (by using [arithmetic coding](https://en.wikipedia.org/wiki/Arithmetic_coding) on the output distribution) while an optimal compressor can be used for prediction (by finding the symbol that compresses best, given the previous history).

In arithmetic coding, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total.

Data compression is an open area of research.

After the deflate algorithm there have been many new discoveries - the most recent one is GOOGLE GUETZLI. The GOOGLE GUETZLI is a JPEG encoder that aims for excellent compression density at high visual quality.

**References:-**

1. https://en.wikipedia.org/
2. <https://github.com/dhruvchadha2212>
3. http://warp.povusers.org/EfficientLZW/index.html
4. https://www2.cs.duke.edu/csed/curious/compression/huffman.html