FILE COMPRESSOR
DESIGN AND ANALYSIS OF ALGORITHMS
MINI-PROJECT REPORT

For the partial fulfillment of completion of degree of B.Tech.(CSE)

Submitted by

KOLLI GOPINADH(RA2011042010121)

P.LOHITH(RA2011042010139)

JAGGA ROYAL (RA2011042010132)

Under the Guidance of

Dr.B.PRABHU KAVIN

AcademicYear:2021-2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Declaration

I, hereby declare that the work presented in this dissertation entitled "FILE COMPRESSOR" has been done by me and my team, and this dissertation embodies my own work

CONTENT

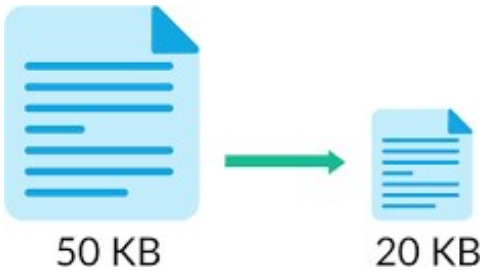
CONTRIBUTION TABLE

SLNO	Name	REGNO	Contribution
1	KOLI GOPINADH	RA2011042010121	Code, algorithm
2	P.LOHITH	RA2011042010139	Preparation of report
3	JAGGA ROYAL	RA2011042010132	Test and analysis of code

ve

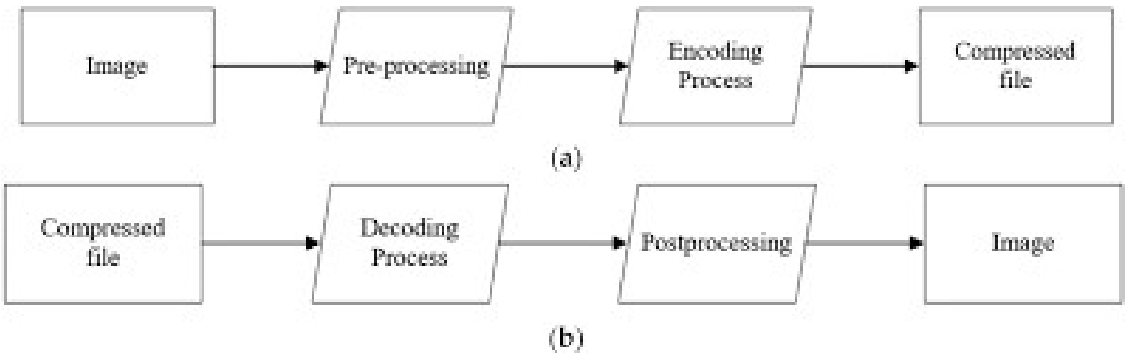
THE PROBLEM :-

The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.



SOLUTION :-

Using the Huffman Coding technique, we can compress the string to a smaller size. Huffman coding first creates a tree using the frequencies of the character and then generates code for each character. Once the data is encoded, it has to be decoded. Decoding is done using the same tree.



ALGORITHM :-

Huffman Coding Algorithm:-

create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
  create a newNode
  extract minimum value from Q and assign it to leftChild of newNode
  extract minimum value from Q and assign it to rightChild of newNode
  calculate the sum of these two minimum values and assign it to the value of newNode
  insert this newNode into the tree
return rootNode
Huffman coding pseudocode:

Compression Technique:-

```
Procedure Huffman(C):          // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
        n = node(C[i])
        Q.push(n)
end for
while Q.size() is not equal to 1
        Z = new node()
        Z.left = x = Q.pop
        Z.right = y = Q.pop
        Z.frequency = x.frequency + y.frequency
        Q.push(Z)
end while
Return Q
```

Decompression Technique:-

```
Procedure HuffmanDecompression(root, S):      // root represents the root of Huffman Tree
n := S.length                                 // S refers to bit-stream to be decompressed
for i := 1 to n
        current = root
        while current.left != NULL and current.right != NULL
                if S[i] is equal to '0'
                        current := current.left
                else
                        current := current.right
                endif
```

```
        i := i+1endwhile
    print current.symbolendfor
```

CODE IMPLEMENTATION :-

```
import heapqimport os

class HuffmanCoding:
        def__init__(self, path):
                self.path = path self.heap = []
                self.codes = {}
                self.reverse_mapping = {}

        class HeapNode:
                def__init__(self, char, freq):
                        self.char = char self.freq = freq
                        self.left = None self.right =
                        None

                # defining comparators less_than and equalsdef__lt
                _____(self, other):
                        return self.freq < other.freq

                def__eq__(self, other):
                        if(other == None):
                                return False
                        if(not isinstance(other, HeapNode)):return False
                        return self.freq == other.freq# functions for

        compression:

        def make_frequency_dict(self, text):frequency = {}
                for character in text:
                        if not character in frequency:
                                frequency[character] = 0
                        frequency[character]    +=    1    return
                frequency
```

```
        def make_heap(self, frequency):
                for key in frequency:
                        node = self.HeapNode(key,
frequency[key])
                        heapq.heappush(self.heap, node)

        def merge_nodes(self):
                while(len(self.heap)>1):
                        node1 = heapq.heappop(self.heap)
                        node2 = heapq.heappop(self.heap)

                        merged = self.HeapNode(None, node1.freq
+ node2.freq)
                        merged.left = node1
                        merged.right = node2

                        heapq.heappush(self.heap, merged)


        def make_codes_helper(self, root, current_code):
                if(root == None):
                        return

                if(root.char != None):
                        self.codes[root.char] = current_code
                        self.reverse_mapping[current_code] =
root.char
                        return

                self.make_codes_helper(root.left, current_code + "0")
                self.make_codes_helper(root.right, current_code +
"1")


        def make_codes(self):
                root = heapq.heappop(self.heap)
                current_code = ""
                self.make_codes_helper(root, current_code)


        def get_encoded_text(self, text):
                encoded_text = ""
                for character in text:
                        encoded_text += self.codes[character]
                return encoded_text
```

```
        def pad_encoded_text(self, encoded_text):
                extra_padding = 8 - len(encoded_text) % 8
                for i in range(extra_padding):
                        encoded_text += "0"

                padded_info = "{0:08b}".format(extra_padding)
                encoded_text = padded_info + encoded_text
                return encoded_text


        def get_byte_array(self, padded_encoded_text):
                if(len(padded_encoded_text) % 8 != 0):
                        print("Encoded text not padded properly")
                        exit(0)

                b = bytearray()
                for i in range(0, len(padded_encoded_text), 8):
                        byte = padded_encoded_text[i:i+8]
                        b.append(int(byte, 2))
                return b


        def compress(self):
                filename, file_extension = os.path.splitext(self.path)
                output_path = filename + ".bin"

                with open(self.path, 'r+') as file, open(output_path,
'wb') as output:
                        text = file.read()
                        text = text.rstrip()

                        frequency = self.make_frequency_dict(text)
                        self.make_heap(frequency)
                        self.merge_nodes()
                        self.make_codes()

                        encoded_text = self.get_encoded_text(text)
                        padded_encoded_text =
self.pad_encoded_text(encoded_text)

                        b =
self.get_byte_array(padded_encoded_text)
                        output.write(bytes(b))

                print("Compressed")
                return output_path
```

```
        """ functions for decompression: """


        def remove_padding(self, padded_encoded_text):
                padded_info = padded_encoded_text[:8]
                extra_padding = int(padded_info, 2)

                padded_encoded_text = padded_encoded_text[8:]
                encoded_text =
padded_encoded_text[:-1*extra_padding]

                return encoded_text

        def decode_text(self, encoded_text):
                current_code = ""
                decoded_text = ""




        def decompress(self, input_path):
                filename, file_extension = os.path.splitext(self.path)
                output_path = filename + "_decompressed" + ".txt"

                with open(input_path, 'rb') as file, open(output_path,
'w') as output:
                        bit_string = ""

                        byte = file.read(1)
                        while(len(byte) > 0):
                                byte = ord(byte)
                                bits = bin(byte)[2:].rjust(8, '0')
                                bit_string += bits
                                byte = file.read(1)

                        encoded_text =
self.remove_padding(bit_string)

                        decompressed_text =
self.decode_text(encoded_text)

                        output.write(decompressed_text)

                print("Decompressed")
                return output_path
```

Time complexity analysis :

The time complexity for encoding each unique character based on its frequency is O(nlog n) . Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n) . Thus the overall complexity is O(nlog n) .

CONSTRAINTS:-

Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.

Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.

TEST CASE :-

Input:

Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output:

Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

RESULT : -

The original representation has 8 bytes(64 bits) and the new representation have only 9 bits, that is 86% smaller than the original.
So **the Huffman Coding turns to be a simple and efficient way to encode data into a short representations without loosing any piece of information**.

REFERENCES : -

Geeks For Geeks, Tutorial Spot.