Visvesvaraya Technological University
Belagavi-590 018, Karnataka

A MINI PROJECT REPORT ON

"Text File And Image Compression"

Mini-Project Report submitted in fulfillment of the requirement for the 6th Semester File Structures Laboratory with Mini-Project [17ISL68]

Bachelor of Engineering
In
Information Science and Engineering

Submitted by

ARJUN S N[1JT17IS007]

Under the Guidance of
Mr.Vadiraja A
Asst.Professor,Department of ISE

Department of Information Science and Engineering
Jyothy Institute of Technology,
Tataguni, Bengaluru-56008

Jyothy Institute of Technology,

Department of Information Science and Engineering
Tataguni, Bengaluru-560082

CERTIFICATE

Certified that the mini project work entitled **"Text File And Image Compression"** carried out by **ARJUN S N [1JT17IS007]** Bonafede student of Jyothy Institute of Technology, in partial fulfillment for the award of **Bachelor of Engineering** in **Information Science and Engineering** Department of the **Visvesvaraya Technological University, Belagavi** during the year **2020-2021**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Degree.

Mr. Vadiraja A
Guide, Asst, Professor
Dept. Of ISE

Dr. Harshvardhan Tiwari
Associate Professor and HoD
Dept. OF ISE

External Viva Examiner

Signature with Date :

-
-

ACKNOWLEDGMENT

The satisfaction that accompanies the successful completion of my project Would be incomplete without mentioning the people who made it possible, I am thankful to **Dr. Gopalakrishna K,** Principal, JIT, Bangalore for Being kind enough to provide us an opportunity to work on a project in this Institution

I am thankful to **Dr. Harshvardhan Tiwari ,** HoD, Dept of ISE for his co-operation and encouragement at all moments of our approach and Whose constant guidance and encouragement crowns all the efforts with Success.

I would greatly mention enthusiastic influence provided by **Mr. Vadiraja A,** Asst. Professor, Dept. of ISE for his ideas and co-operation Showed on us during our venture and making this project a great success.

Finally, it's pleasure and happiness to the friendly co-operation showed by all The staff members of Information Science Department, JIT.

ARJUN S N
1JT17IS007

ABSTRACT

Data compression is also called as source coding. It is the process of encoding information using fewer bits than an uncoded representation is also making a use of specific encoding schemes. Compression is a technology for reducing the quantity of data used to represent any content without excessively reducing the quality of the picture. It also reduces the number of bits required to store and/or transmit digital media. Compression is a technique that makes storing easier for large amount of data

The File Compression mini project is to design a compression program which significantly reduces the size of a file so that it can be easily shared over and save storage capacity, speed up file transfer, and decrease costs for storage hardware and network bandwidth. Each byte of the file will be compressed using encoding algorithm and can be decompressed to original file using decode algorithm.

In this project, will be implementing lossless and lossy compression methods where lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression so it is good for text compression. And Lossy compression reduces bits by removing unnecessary or less important information so it is good for image compression.

This program will be implemented to compress text files using **Huffman coding** algorithm and images using Basic inbuilt (RESIZE) function of Pillow which is lossy compression .

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

INTRODUCTION

1.1 Introduction to File Structures

In simple terms, a file is a collection of data stored on mass storage (e.g., disk or tape).But there is one important distinction that must be made at the outset when discussing file structures. And that is the difference between the logical and physical organization of the data.

On the whole a file structure will specify the logical structure of the data, that is the relationships that will exist between data items independently of the way in which these relationships may actually be realized within any computer. It is this logical aspect that we will concentrate on. The physical organization is much more concerned with optimizing the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of physical store there will be a number of units of the logical structure (probably records) to be stored in it.

For example, if we were to store a tree structure on a magnetic disk, the physical organization would be concerned with the best way of packing the nodes of the tree on the disk given the access characteristics of the disk.

Like all subjects in computer science the terminology of file structures has evolved higgledy-piggledy without much concern for consistency, ambiguity, or whether it was possible to make the kind of distinctions that were important.

It was only much later that the need for a well-defined, unambiguous language to describe file structures became apparent. In particular, there arose a need to communicate ideas about file structures without getting bogged down by hardware consideration

1.2 Introduction to File System

In computing, a file system or file system controls how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified.

Taking its name from the way paper-based information systems are named, each groups of data is called a “file”. The structure and logic rules used to manage the groups of information and their names is called a “file system”. There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more. Some file systems have been designed to be used for specific applications. For example, the ISO 9660 file system is designed specifically for optical discs.

File systems can be used on numerous different types of storage devices that use different kinds of media. The most common storage device in use today is a hard disk drive. Other kinds of media that are used include flash memory, magnetic tapes, and optical discs. In some cases, such as with tmpfs, the computer's main memory (random-access memory, RAM) is used to create a temporary file system for short-term use.

1.3 Data Compression

With the advancement in technology, multimedia content of digital information is increasing day by day, which mainly comprises of data. Hence storage and transmission of these files require a large memory space and bandwidth. The solution is to reduce the storage space required for these files while maintaining acceptable File quality.

How compression works

Compression is performed by a program that uses a formula or algorithm to determine how to shrink the size of the data. For instance, an algorithm may represent a string of bits -- or 0s and 1s -- with a smaller string of 0s and 1s by using a dictionary for the conversion between them, or the formula may insert a reference or pointer to a string of 0s and 1s.

Text compression can be as simple as removing all unneeded characters, inserting a single repeat character to indicate a string of repeated characters and substituting a smaller bit string for a frequently occurring bit string. Data compression can reduce a text file to 50% or a significantly higher percentage of its original size.

For data transmission, compression can be performed on the data content or on the entire transmission unit, including header data. When information is sent or received via the internet, larger files, either singly or with others as part of an archive file, may be transmitted in a ZIP, GZIP or other compressed Format

Huffman Coding Compression technique

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1 internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The compressor algorithm builds a string translation table from the text being compressed. The string translation table maps fixed-length codes (usually 12-bit) to strings. The string table is initialized with all single-character strings (256 entries in the case of 8-bit characters). As the compressor character-serially examines the text, it stores every unique two-character string into the table as a code/character concatenation, with the code mapping to the corresponding first character.

As each two-character string is stored, the first character is outputted. Whenever a previously-encountered string is read from the input, the longest such previously-encountered string is determined, and then the code for this string concatenated with the extension character (the next character in the input) is stored in the table. The code for this longest previously-encountered string is outputted and the extension character is used as the beginning of the next string

Huffman Codding Decompression technique

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using canonical encoding, the compression model can be precisely reconstructed with just B2^{B} bits of information .Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are

possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).The decompressor algorithm only requires the compressed text as an input, since it can build an identical string table from the compressed text as it is recreating the original text.

Pillow Image Module For image compression

PIL is the Python Imaging Library which provides the python interpreter with image editing capabilities. The Image module provides a class with the same name which is used to represent a PIL image. The module also provides a number of factory functions, including functions to load images from files, and to create new images.
Image.resize() Returns a resized copy of this image.

size – The requested size in pixels, as a 2-tuple: (width, height).

resample – An optional resampling filter. This can be one of PIL.Image.NEAREST (use nearest neighbour), PIL.Image.BILINEAR (linear interpolation), PIL.Image.BICUBIC (cubic spline interpolation), or PIL.Image.LANCZOS (a high-quality downsampling filter). If omitted, or if the image has mode “1” or “P”, it is set PIL.Image.NEAREST.

For this Project I will be using ANTIALIAS Filter of Image module for compressing the image .It is a inbuilt filter which is very good for optimizing images

CHAPTER 2

REQUIREMENT ANALYSIS AND DESIGN

REQUIREMENT ANALYSIS AND DESIGN

2.1 Domain Understanding

The Main Objective of this Project is to Compress The Given Text File or Image File. The outcome is it gives a Compressed File in an Efficient and Friendly in understandable GUI.

2.2 Classification of Requirements

System Requirements

- Processors: Intel Atom® processor or Intel® Core™ i3 processor
- Disk space: 100 GB
- Ram: 4 GB
- Operating systems: Windows* 7 or later, macOS, and Linux
- Python* versions: 3.6.X or Higher

Software and Hardware Requirements

- Basic Hardware Requirement:4gb Ram,100gb Rom, dual core processor
- Python IDE
- Pillow,Pyqt5 packages
- Programming Languages: Front End – Pyqt5

  Back End – Python

2.3 System Analysis

When the program is executed, it loads a simple Gui which primarily asks the user to press Which type of data to be compressed Text or Image. If user choose text then there will be another choice asking single text file or multiple text file, If user choose single text file then they get option the Compression and Decompression whatever they want to access. If the user chooses the Compression button, then it asks to input the TextFile which they want to compress that. After the TextFile is obtained, the entire File is viewed for verification of Size of the file with name and location.

When user press Compress button it starts to compress the TextFile. After the completion of compressing, the user is given information about compression such as time taken, Compressed Textfile Ratio.

When user press Decompression button it st2arts to Decompress the TextFile and it takes only .bin File.After the completion of decompressing, the user is given information about decompression such as time taken.

If user chose multiple text file option then it executes same as single file but only change is it takes whole directory of the TextFiles present as Input and user will be also asked in which directory to save it.

For image choice is clicked then user is asked for single or multiple images ,if single image is clicked then user is asked for which image needs to be compressed and where to save it. Same thing go for multiple images where whole directory consisting of images and also user get to choose there destination directory.

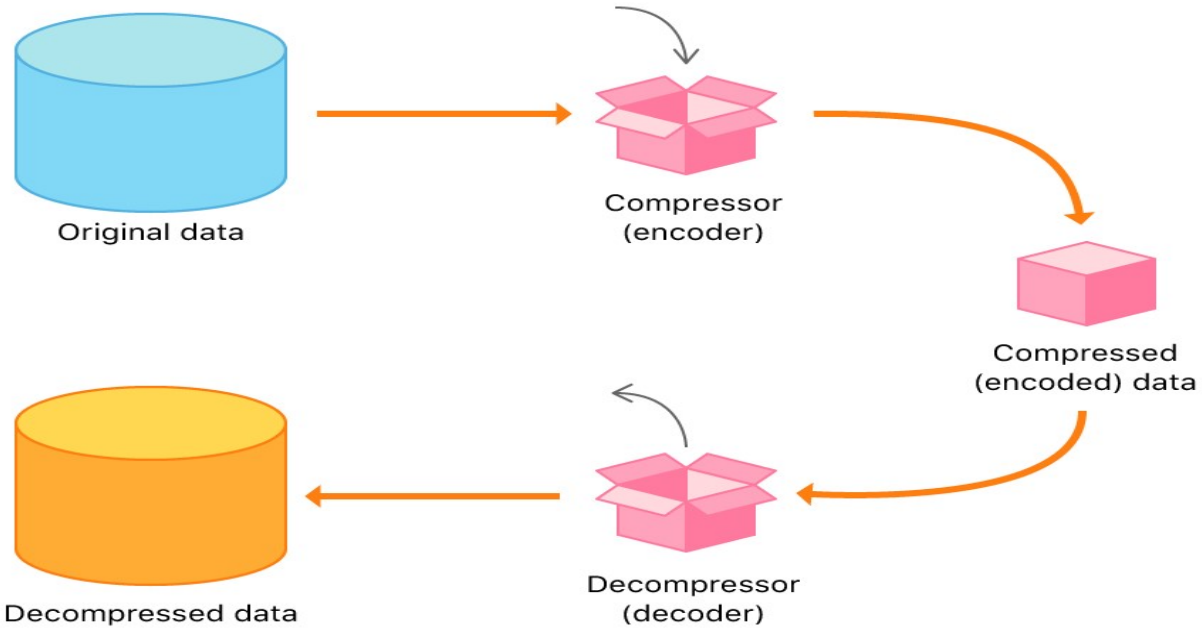


Fig:2.3.1

In Fig 2.3.1 we can see how the Text file compression and decompression takes place and it is lossless method so no data is lost

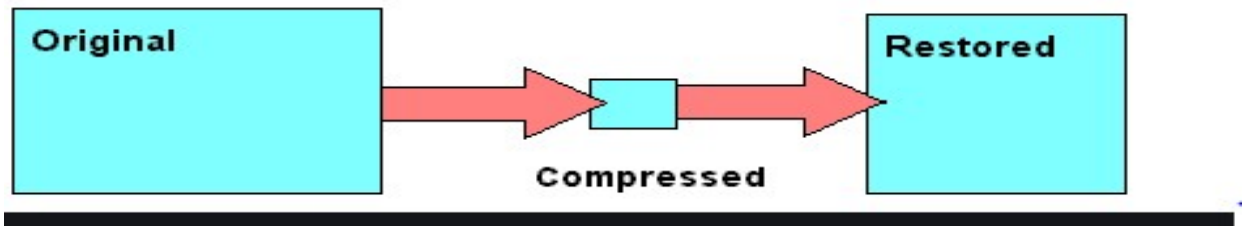


Fig:2.3.2

In Fig:2.3.2 we can see how image compression works in top level and as we can see in image the restored data is not same as the original as we are using lossy method we can't restore it exactly as original

CHAPTER 3
IMPLEMENTATION

IMPLEMENTATION

The Text File Compression Using Huffman Coding Algorithm is Implemented Through Two Types of Tehnique

- HUFFMAN CODING COMPRESSION
- HUFFMAN CODING DECOMPRESSION

- **HUFFMAN CODING COMPRESSION**

```
Begin

   define a node with character, frequency, left and right child of the
node for Huffman tree.

   create a list 'freq' to store frequency of each character,
initially, all are 0

   for each character c in the string do

      increase the frequency for character ch in freq list.

   done


   for all type of character ch do

      if the frequency of ch is non zero then

         add ch and its frequency as a node of priority queue Q.

   done

   while Q is not empty do

      remove item from Q and assign it to left child of node

      remove item from Q and assign to the right child of node

      traverse the node to find the assigned code

   done

End
```

The Above Algorithm Shows How The Huffman CodingAlgorithm Compresses The Given File.

- **HUFFMAN CODING DECOMPRESSION**

Algorithm for Huffman Coding Decompression

```
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+1
   endwhile
   print current.symbol
endfor
```

The Above Algorithm Shows How The Huffman Coding Algorithm DeCompresses The Given File.

Implementing/Programming Huffman Coding

In this section we'll see the basic programming steps in implementing Huffman coding. There are two parts to an implementation: a compression program and a decompression program. We'll assume these are separate programs, but they actually have many common functions. For this set of programming exercises/assignment, you will deal mainly with decompression. However, to do decompression, we have to understand compression.

The Compression Program

To compress a file (sequence of characters) you need a table of bit encodings, i.e., a table giving a sequence of bits used to encode each character. This table is constructed from a coding tree using root-to-leaf paths to generate the bit sequence that encodes each character.

A compressed file is obtained using the following top-level steps. These steps will be developed further into sub-steps, and you'll eventually implement a program based on these ideas and sub-steps.

1. Build a Huffman coding tree based on the number of occurrences of each ASCII character in the file. Build a table of Huffman codes for all ASCII characters that appear in the file.
2. Read the file to be compressed (the plain file) and process one character at a time. To process each character find the bit sequence that encodes the character using the table built in the previous step and write this bit sequence to the compressed file.

The main challenge here is that when you encode an ASCII character read from the file, the code is typically shorter than 8 bits. However, most systems allow you to write to a file a byte or 8 bits at a time. It becomes necessary for you to accumulate the codes for a few ASCII characters before you write an 8-bit character to the output file.

To compress the string "streets are stone stars are not" for example, we read from the string one character at a time. The code for 's' is 111, we cannot write to the file yet. We read the next character 't', whose code is 00. Again, we cannot write to the output file because the total number of bits is only 5. Now, we read the next character 'r', whose code is 011. The compression program can now print a character of bit pattern 11100011 to the output file.

Again, a reminder that we use the bit ordering in a byte to be one where the most significant bit is on the left and the least significant bit is on the right.

Continuing with the process, the program would have to read 'eet', whose combined code is '11011000', before printing the 8-bit combined code to the output file. When the program reads the next three characters 's a', the combined code is '111101010'. As the combined code has one more than 8 bits, the program should print the first 8 bits to the output file, and combine the remaining bit with the next characters. You may also encounter codes that are longer than 8 bits. You would have to split such codes into 8-bit bytes.

What happens when you reach the end of the file, and you have not accummulated 8 bits for printing. All you have to do is pad the accummulated bits with enough 0's and print it to the output file.

We will show you the header information that a compression program that I have written provides at the beginning of the compressed file so that the decompression program can correctly decode the compressed file.

```
class HuffmanCoding:
    def __init__(self):...

    # make frequency dictionaries with sorted value from low to high
    def make_frequency_dict(self, text):
        counted = dict(collections.Counter(text))
        sort = collections.OrderedDict(sorted(counted.items(),key=operator.itemgetter(1), reverse=False))
        #print(sort)
        return sort

    # make a heap queue from node
    def make_heap_node(self, freq_dict):
        for key in freq_dict:
            anode = HeapNode(key, freq_dict[key])
            self.heap.append(anode)

    # build tree
    def merge_nodes(self):
        while len(self.heap) > 1:
            node1 = heapq.heappop(self.heap)
            node2 = heapq.heappop(self.heap)
            merge = HeapNode(None, node1.freq + node2.freq)
            merge.left = node1
            merge.right = node2
            heapq.heappush(self.heap, merge)
```

Fig:3.1-Buliding Tree

Header Information

You must store some initial information in the compressed file that will be used by the decompression/unhuffing program. Essentially, you must store the tree that is used to compress the original file. This is because the decompression program needs this exact same tree in order to decode the data. We will refer to the topology as the header information. At the beginning of the compressed, there are three unsigned integers:

- The total number of characters in the compressed file, as a 4-byte unsigned integer.

- The total number of characters storing the header information, i.e., the topology of the Huffman coding tree, as a 4-byte unsigned integer.

- The total number of characters in the original uncompressed file, as a 4-byte unsigned integer.

```
# Encoding Starts here
def encode_helper(self, root, current_code):
    if root is None:
        return

    if root.char is not None:
        self.codes[root.char] = current_code
        return

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

def encode(self):
    root = heapq.heappop(self.heap)
    current_code = ""
    self.encode_helper(root, current_code)

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

def pad_encoded_text(self, encoded_text):
    # get the extra padding of encoded text
    extra_padding = 8 - len(encoded_text) % 8
    for i in range(extra_padding):
        encoded_text += "0"
    # merge the "info" of extra padding in "string/bit" with encoded text
    # so we know how to truncate it later
    padded_info = "{0:08b}".format(extra_padding)
    new_text = padded_info + encoded_text

    return new_text

def to_byte_array(self, padded_encoded_text):
    if len(padded_encoded_text) % 8 != 0:
        print('not padded properly')
        exit(0)
    b = bytearray()
    for i in range(
            0, len(padded_encoded_text), 8):  # loop every 8 character
        byte = padded_encoded_text[i:i + 8]
        b.append(int(byte, 2))  # base 2
    return b
```

Fig:3.2-Encoding

Following the three unsigned integers, we store the header information or the topology of the Huffman coding tree, and then followed by the encoding of the original text using Huffman codes. We have described the encoding of the original text in the

earlier sections. Now, we focus on how the topology of the Huffman coding tree is stored.

To store the tree at the beginning of the file, we use a post-order traversal, writing each node visited. When you encounter a leaf node, you write a 1 followed by the ASCII character of the leaf node. When you encounter a non-leaf node, you write a 0. To indicate the end of the Huffman coding tree, we write another 0.

```
def compress(self, filename, path):
    start = time.time()
    file_text = open(filename, 'r')
    lipsum = file_text.read()
    file_text.close()

    freq = self.make_frequency_dict(lipsum)
    self.make_heap_node(freq)
    self.merge_nodes()
    self.encode()
    encoded_text = self.get_encoded_text(lipsum)
    padded_encoded_text = self.pad_encoded_text(encoded_text)
    byte_array_huff = self.to_byte_array(padded_encoded_text)

    # write header
    filename_split = filename.split('.')
    fname = str(filename_split[0]).split("/")
    js = open(path+"/"+fname[-1] + "_compressed_" + filename_split[1] + "_.bin", 'wb')
    strcode = str(self.codes)
    js.write(strcode.encode())
    js.close()

    # append new line for separation
    append = open(path+"/"+fname[-1] + "_compressed_" + filename_split[1] + "_.bin", 'a')
    append.write('\n')
    append.close()

    # append the rest of the "byte array"
    f = open(path+"/"+fname[-1] + "_compressed_" + filename_split[1] + "_.bin", 'ab')
    f.write(bytes(byte_array_huff))
    f.close()

    # Time and Compression percentage
    get_original_filesize = os.path.getsize(filename)
    get_compressed_filesize = os.path.getsize(path +"/"+
                              fname[-1] + "_compressed_" + filename_split[1] + "_.bin")
    percentage = (get_compressed_filesize / get_original_filesize) * 100
    end = time.time()
    self.msg = 'Compression Done! with Ratio ' + str(round(percentage, 3)) + "% And time take is " + str(
        round((end - start),
              3)) + "s"
```

Fig:3.3-Compression Function

The header information can be made more economical if we use bits 0 and 1 to distinguish between non-leaf and leaf nodes, and also to indicate the end of a topology. In these two examples, there will be a total of 10 bytes (8 bytes for the leaf nodes and 2 bytes for all the 0's and 1's). The challenge here is that both the compression and decompression programs would have to handle bits instead of characters. For example, in the bit-based approach, the first 16 bits (or the first 2 bytes) of the header information for encoding the string "streets are stone stars are not" are 10111010 01011000 (note that the space in the bit stream is introduced for better clarity). The ASCII coding for 't' straddles the two bytes, 7 bits in the first byte and 1 bit in the second byte. The second most significant bit in the second byte is a 1, indicating that the next 8 bits is an ASCII character, of which the the most significant 6 bits of the character 'a' is contained in the least significant 6 bits of the second byte.

In the bit-based representation of the Huffman coding tree, the last byte may not contain 8 bits. In this case, we again pad it with 0 bits. Consider the case where the input file uses nine distinct ASCII characters. The number of bits required to represent the Huffman coding tree is $9\times8 + 9\times2 = 90$ bits, which can represented by 12 bytes. In other words, the last byte should contain only two useful bits. The 12 bytes are followed by the encoded text.

The Decompression or Dehuffing Program

Given a compressed file, which contains the header information, followed by a bit stream corresponding to the encoding of the original file, the decompression program should perform the following tasks:

1. Build a Huffman coding tree based on the header information. You would probably need the second unsigned integer stored at the beginning of the compressed file to help with the construction.
2. Read bit-by-bit from the compressed file (starting at the location after the header information). Use the read bit to traverse the Huffman coding tree (0 for left and 1 for right), starting from the root node. When the program reaches a leaf node, print the corresponding ASCII character to the output file. Starts from the root node of the Huffman coding tree when the next bit is read. The program terminates when the number of characters decoded matches the number of characters stored as the third unsigned integer at the beginning of the compressed file.

To construct a Huffman coding tree from the header information, we make use of a stack. When a 1 (bit or character depending on whether we are dealing with a bit-based or character-based representation) is read, we read the next byte and push the corresponding ASCII character onto the stack. When a 0 (bit or character) is read, if the stack contains only one element, we have constructed the entire Huffman coding tree. Otherwise, there must be more than one element in the stack. We create a new node, and pop the top two elements off the stack. We make the first element off the stack the right child of the new node, and the second element off the stack the left child of the new node. After that, we push the newly created node onto the stack.

```
def decompress(self, compressedfile, path):
    start = time.time()
    filename_split = compressedfile.split('_')
    fname=str(filename_split[0]).split("/")
    # get "header"
    header = open(compressedfile, 'rb').readline().decode()
    # header as object literal
    header = ast.literal_eval(header)
    # reverse mapping for better performance
    self.reverse_mapping = {v: k for k, v in header.items()}
    # get body
    f = open(compressedfile, 'rb')
    # get "body" as list.  [1:] because header
    body = f.readlines()[1:]
    f.close()

    bit_string = ""
    # merge list "body"
    # flattened the byte array
    join_body = [item for sub in body for item in sub]
    for i in join_body:
        bit_string += "{0:08b}".format(i)

    encoded_text = self.remove_padding(bit_string)
    # decompress start here
    current_code = ""
    decoded_text = ""
    for bit in encoded_text:
        current_code += bit
        if current_code in self.reverse_mapping:
            decoded_text += self.reverse_mapping[current_code]
            current_code = ""
    write = open(path + "/" + fname[-1] + "_decompressed." + filename_split[2], 'w')
    write.writelines(decoded_text)
    write.close()
    end = time.time()
    self.msg = 'Decompression Done! and Time taken is ' + str(round((end - start), 3)) + "s"
```

Fig:3.4-Decompression

A challenge is in the reading of a bit from the compressed file. Again, reading from a file occurs at the byte-level. In the decompression program, you have to read an 8-bit ASCII character that straddle two bytes in the compressed file when you want to construct a Huffman coding tree. You also have to read one bit at a time to determine whether you are dealing with a leaf node or a non-leaf node. Of course, you also have to read one bit at a time when you want to decode the compressed file. A strategy is to always read a byte from the compressed file. You also maintain an index to indicate where you are in that byte. For example, when you read in a character, the index should be pointing to the most significant bit (i.e., the 7th bit, assuming that the least significant bit is the 0th bit). After you have processed one bit, the index should be decremented by 1.

- **The Image Compression**

The Image Compression Using PIL (pillow) Image module is Implemented using antialias filter

Basically we will be taking image from the user as the input and we will use pillow module to open the image and get few details of the image .After that we will get the width of the image ,we can decrease width if we don't need good quality and user can choose it ,but the width size should not be greater than the original width of the image.

After doing that now we need to find height such that aspect ratio remains same after doing that we need to just call resize method which takes width ,height and filter method we will be using antialias filter which is very good for quality but it lacks performance. After doing that the image is saved to the destination path given by the user.

For multiple images we will be using os.listdir method which takes directory path and creates a list of image files present in that directory and we access each one and pass it to above image compression function which compresses images and returns compressed images each compressed image is save to the destination directory given by user

Algorithm for Image compression

Step 1: Take input image from user

Step 2: Ask user input where to save the compressed image

Step3: open image using Image module and get the width of the image

Step4: width percent = original width of image / img.size

Step5: height size = img.size[1] * wpercent

Step6: image.resize((original width, height size), PIL.Image.ANTIALIAS)

Step7:Save image to the destination path

```
def compress_img(old_img, new_img, mywidth):
    img = Image.open(old_img)
    wpercent = (mywidth / float(img.size[0]))
    hsize = int((float(img.size[1]) * float(wpercent)))
    img = img.resize((mywidth, hsize), PIL.Image.ANTIALIAS)
    img.save(new_img)
```

Fig3.5-Image compression Function

```
def mi_com(self):
    spath = self.Multiple_file_img_path.text()
    dpath = self.Multiple_file_img_path1.text()
    if spath=="":
        self.message_text.setText("Please Enter the source path")
        self.message_text.exec()
        exit()
    if dpath=="":
        self.message_text.setText("Please Enter destination path")
        self.message_text.exec()
        exit()
    files = os.listdir(spath)
    start = time.time()
    for file in files:
        old_img = spath + "/" + file
        new_img = dpath + "/" + file
        img = Image.open(old_img)
        self.image_width = img.width
        im.compress_img(old_img, new_img, int(self.image_width/self.i))
    end = time.time()
    self.message_text.setText("Compression Done! in " + str(round((end - start), 3)) + "s")
    self.message_text.exec()
```

Fig3.6-Multiple Image Compression

- **USER INTERFACE**

We are going to implement GUI using pyqt5. Qt is set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems. These include location and positioning services, multimedia, NFC and Bluetooth connectivity, a Chromium based web browser, as well as traditional UI development.

PyQt5 is a comprehensive set of Python bindings for Qt v5. It is implemented as more than 35 extension modules and enables Python to be used as an alternative application development language to C++ on all supported platforms including iOS and Android.

PyQt5 may also be embedded in C++ based applications to allow users of those applications to configure or enhance the functionality of those applications.

- **Implementation of pyqt5**

First step was to define init method in which we can set the size of window, and also for this project I have made the window static size so it can't be resized ,I have assigned size of my window as 400px width 500px height.

Second step is to set the stylesheet for our project so we use setStylesheet method to set it so we can add sqss styles to our pyqt5 widgets .

```
class App(QWidget):

    def __init__(self):
        super().__init__()
        self.title = 'Compressor'
        self.left = 400
        self.top = 50
        self.width = 400
        self.height = 500
        self.image_width = 0
        self.i = 1
        self.setFixedSize(self.width, self.height)
        self.setObjectName("main_window")
        self.setStyleSheet(design.stylesheet)
        self.initUIO
```

Fig3.6-init function of GUI (pyqt5)

Third step is now to add two frames to our main window in first frame we set text as Text File and for the next frame we set text as Images and also we mention the co-ordinates to where we want to place the frames in the window.

```
# ------------------------heading------------------------------------
self.headr = QFrame(self) # frame=Text
self.headr.setObjectName("Header")
self.headr.move(25, 20)
self.headr_heading = QLabel(self.headr)
self.headr_heading.setObjectName("Header_Text")
self.headr_heading.setText("COMPRESSOR")

# ------------------------main_window------------------------------------
self.frame = QFrame(self) # frame=Text
self.frame.setObjectName("Text_button")
self.frame.move(50, 120)
self.frame.mousePressEvent = self.frame_clicked
self.frame_heading = QLabel(self.frame)
self.frame_heading.setObjectName("Heading")
self.frame_heading.move(108, 40)
self.frame_heading.setText("TEXT")

self.frame1 = QFrame(self) # frame1==Image
self.frame1.setObjectName("Text_button")
self.frame1.move(50, 295)
self.frame1.mousePressEvent = self.frame1_clicked
self.frame1_heading = QLabel(self.frame1)
self.frame1_heading.setObjectName("Heading")
self.frame1_heading.move(95, 40)
self.frame1_heading.setText("IMAGE")
```

Fig3.7-Adding Frames to our our Main window

Fourth step is assign click event listener to the frames when user clicks on text he should go to another window which consist of two frames single file or multi files for both text as well as image.

```
# ------------------frame-clked-expand-----------------------------------------
self.frame_expanded = QFrame(self) # frame_expanded=single Text file
self.frame_expanded.setObjectName("Text_button")
self.frame_expanded.move(50, 120)
self.frame_expanded.setVisible(False)
self.frame_expanded.mousePressEvent = self.frame_expanded_clicked
self.frame_expanded_heading = QLabel(self.frame_expanded)
self.frame_expanded_heading.setVisible(False)
self.frame_expanded_heading.move(25, 40)
self.frame_expanded_heading.setObjectName("Heading")
self.frame_expanded_heading.setText("Single File")

self.frame_expanded1 = QFrame(self)  # frame_expanded1= multiple text file
self.frame_expanded1.setObjectName("Text_button")
self.frame_expanded1.move(50, 295)
self.frame_expanded1.setVisible(False)
self.frame_expanded1.mousePressEvent = self.frame_expanded1_clicked
self.frame_expanded1_heading = QLabel(self.frame_expanded1)
self.frame_expanded1_heading.setVisible(False)
self.frame_expanded1_heading.move(25, 40)
self.frame_expanded1_heading.setObjectName("Heading")
self.frame_expanded1_heading.setText("Multi  File")
self.back_arrow_frame_t = QFrame(self)  # frame1_expanded=single Img file
self.back_arrow_frame_t.setObjectName("Back_arrow_Frame")
self.back_arrow_frame_t.move(5, 90)
self.back_arrow_frame_t.setVisible(False)
self.back_arrow_frame1_t = QLabel(self.back_arrow_frame_t)
self.back_arrow_frame1_t.move(0, 0)
self.back_arrow_frame1_t.setObjectName("Frame_Backarrow")
self.back_arrow_frame1_t.setTextFormat(Qt.RichText)
self.back_arrow_frame1_t.setText("&#8592;")
self.back_arrow_frame1_t.mousePressEvent = self.back_arrow_frame1_t_clicked
```

Fig:3.8-Text file Clicked GUI

Fifth step is add file dialog to both text and image option and take the path from the file dialog and we can pass it to compression or decompression function based on the user input

```
# ------------------functions-----------------------------------------

def frame_clicked(self, event):
    self.back_arrow_frame_t.setVisible(True)
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.frame1_expanded.setVisible(False)
    self.frame_expanded.setVisible(True)
    self.frame_expanded_heading.setVisible(True)
    self.frame_expanded1.setVisible(True)
    self.frame_expanded1_heading.setVisible(True)
    print("frame-clicked")

def frame1_clicked(self, event):
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.frame_expanded.setVisible(False)
    self.frame1_expanded.setVisible(True)
    self.frame1_expanded_heading.setVisible(True)
    self.frame1_expanded1.setVisible(True)
    self.frame1_expanded1_heading.setVisible(True)
    self.back_arrow_frame.setVisible(True)
```

Fig3.9:Frame Clicked Functions

Sixth step in this step we add back button to all other windows expect main window and add mouse event accordingly

```
def back_arrow_s_clicked(self, event):
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.Single_text_clicked_frame.setVisible(False)
    self.frame_expanded.setVisible(True)
    self.frame_expanded_heading.setVisible(True)
    self.frame_expanded1.setVisible(True)
    self.frame_expanded1_heading.setVisible(True)
    self.back_arrow_frame_t.setVisible(True)
    print("clicked")

def back_arrow_m_clicked(self, event):
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.Multiple_text_clicked_frame.setVisible(False)
    self.frame_expanded.setVisible(True)
    self.frame_expanded_heading.setVisible(True)
    self.frame_expanded1.setVisible(True)
    self.frame_expanded1_heading.setVisible(True)
    self.back_arrow_frame_t.setVisible(True)
    print("clicked")

def back_arrow_si_clicked(self, event):
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.Single_img_clicked_frame.setVisible(False)
    self.frame1_expanded.setVisible(True)
    self.frame1_expanded_heading.setVisible(True)
    self.frame1_expanded1.setVisible(True)
    self.frame1_expanded1_heading.setVisible(True)
    print("clicked")
    self.back_arrow_frame.setVisible(True)

def back_arrow_mi_clicked(self, event):
    self.frame.setVisible(False)
    self.frame1.setVisible(False)
    self.Multiple_img_clicked_frame.setVisible(False)
    self.frame1_expanded.setVisible(True)
    self.frame1_expanded_heading.setVisible(True)
    self.frame1_expanded1.setVisible(True)
    self.frame1_expanded1_heading.setVisible(True)
    print("clicked")
    self.back_arrow_frame.setVisible(True)
```

Fig:3.10-Back arrow function

Seventh step is to assign functions to buttons like compression method to compress button like that for decompress button also And set message box so after the execution of the function we can see the output image through message box also we can obtain time taken and compress ration by setting it to the message box

```
def st_com(self):
    path = self.single_file_text_path.text()
    des = self.single_file_text_path1.text()
    if path == "":
        self.message_text.setText("Please Enter the source path")
        self.message_text.exec()
        exit()
    if des == "":
        self.message_text.setText("Please Enter destination path")
        self.message_text.exec()
        exit()
    h = HuffmanCoding()
    h.compress(path, des)
    self.message_text.setText(h.msg)
    self.message_text.exec()
    self.single_file_text_path.clear()
    self.single_file_text_path1.clear()
```

Fig 3.11-Compression function with message box

CHAPTER 4
RESULT AND ANALYSIS

RESULT AND ANALYSIS

Figure 5.1:-HOME SCREEN

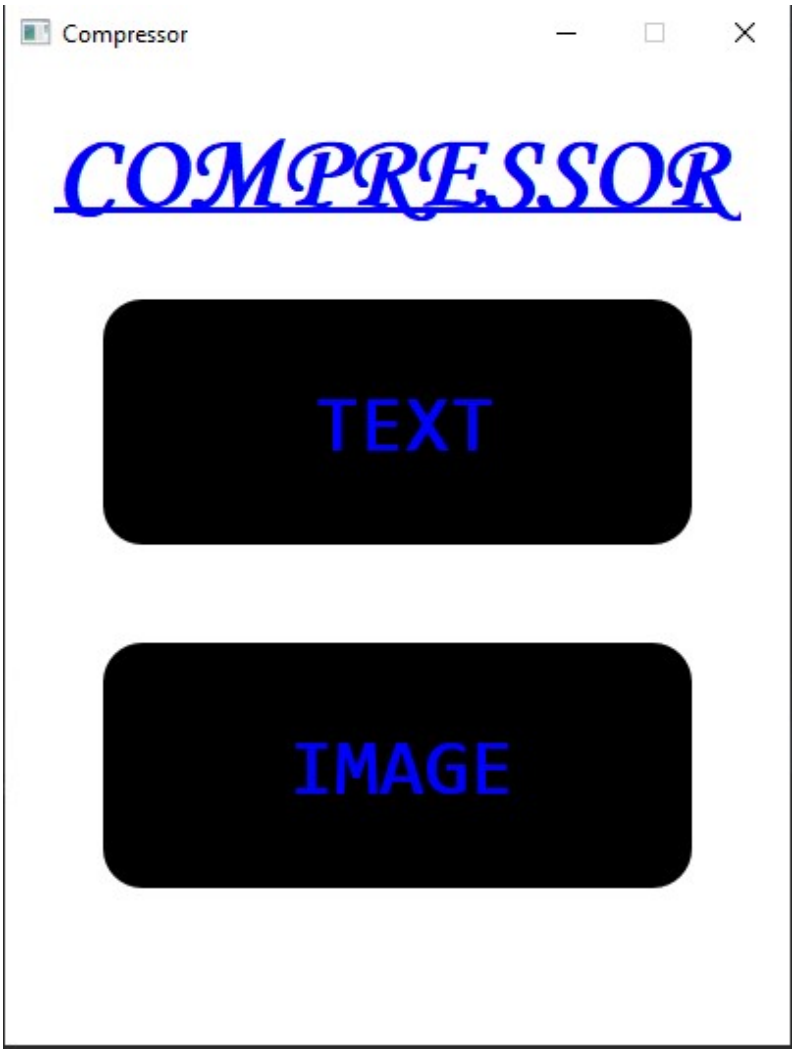


Fig: 5.1

The Above Figure Is the User Interface Home Screen In which it Contains Two Frames Text and Image and The Title Compressor

Figure 5.2:-Text options

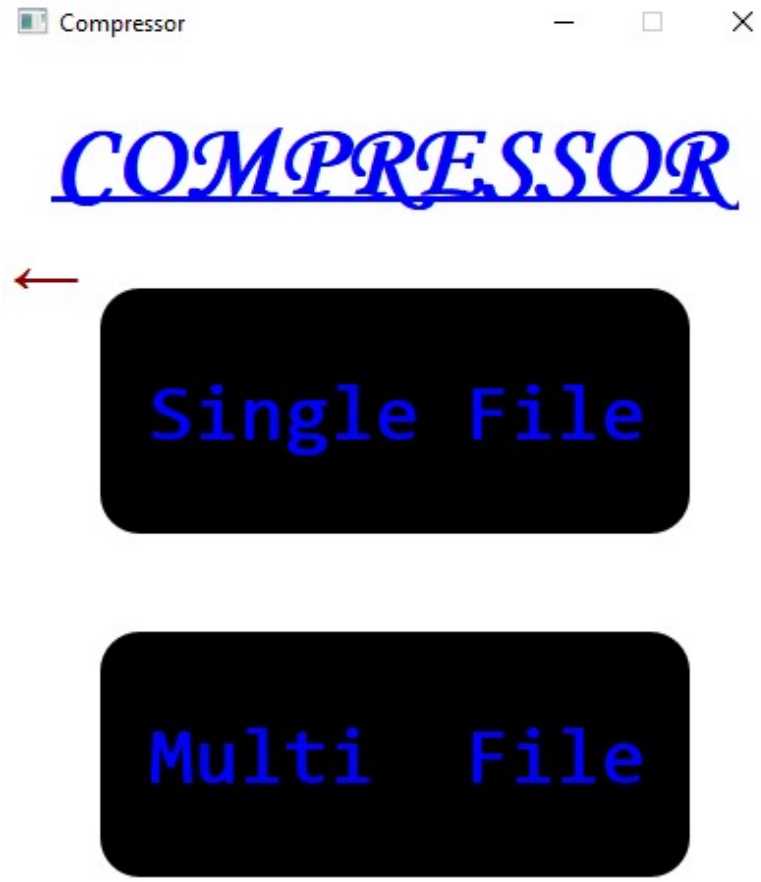


Fig:5.2

The Text Field In which User has to Choose whether they want to single file or multiple file in Which They want to Compress.

Figure 5.3:-Single Text file Compress window

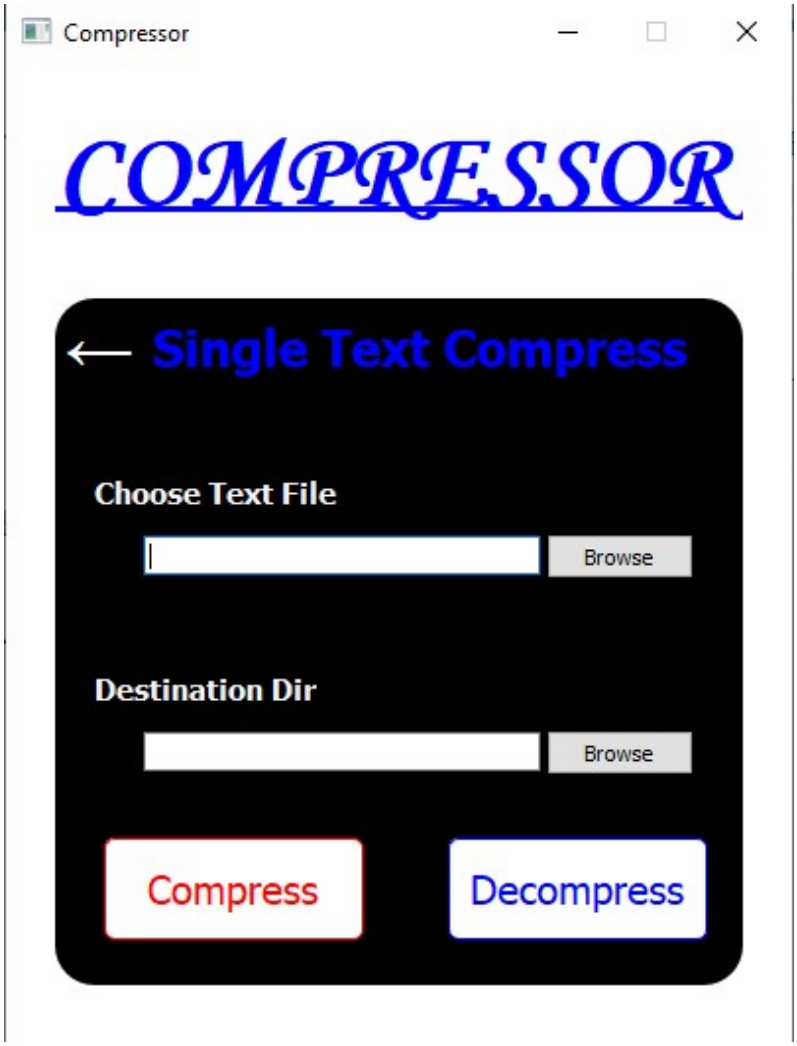


Fig 5.3

Single text compress window where user is given option to select there image by clicking on browse button and also user has to specific the destination directory

Figure 5.4:-File dialog for selecting text file

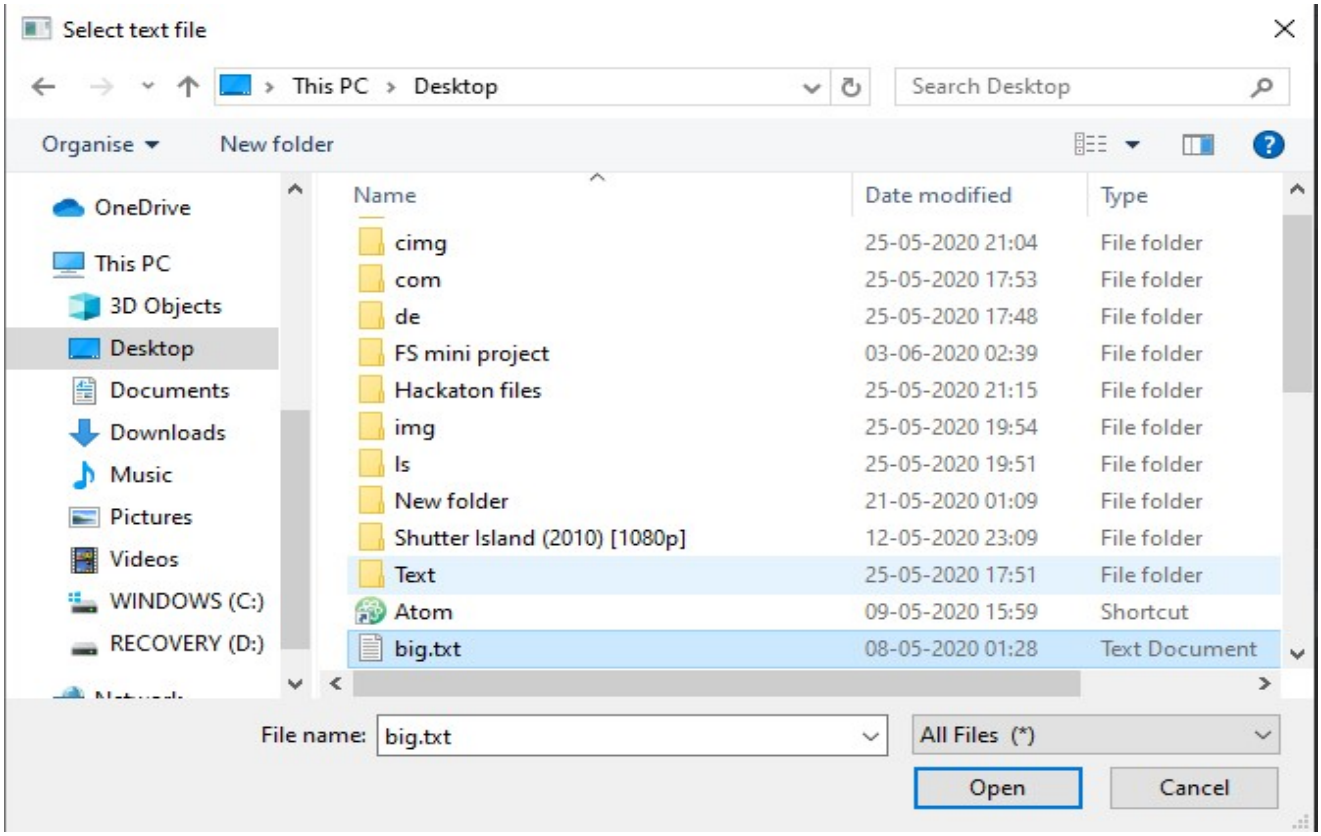


Fig:5.4

Figure 5.5-Fail dialog for selecting destination directory

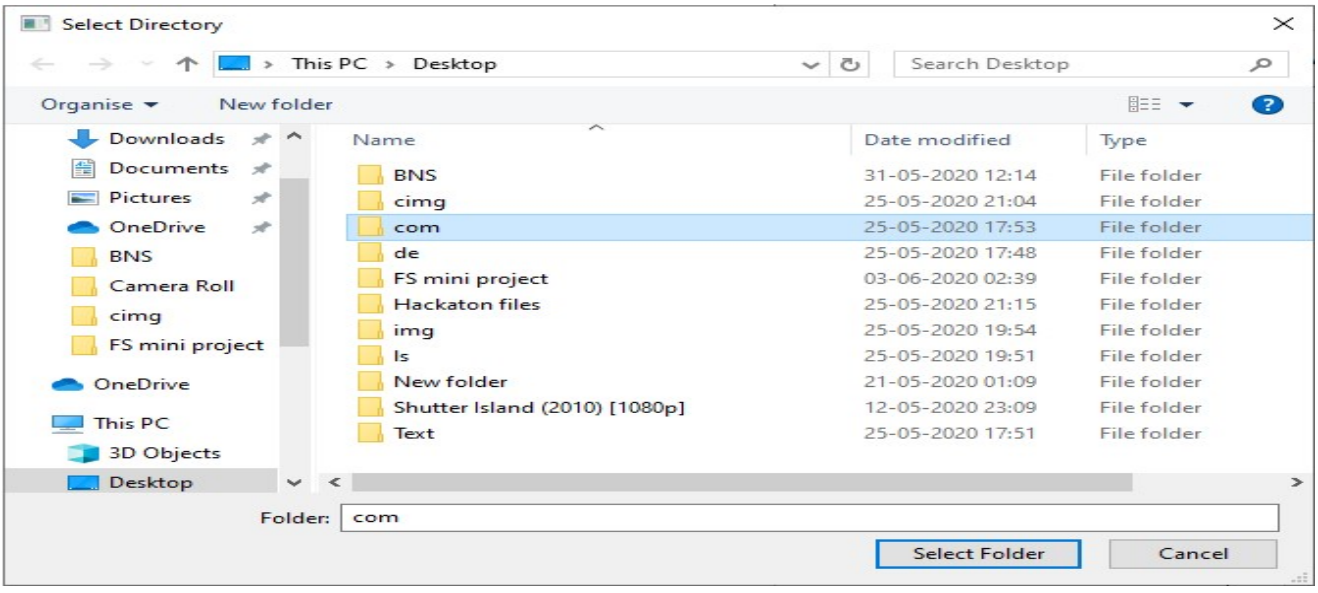


Fig:5.5

Figure 5.6:-Compression Output

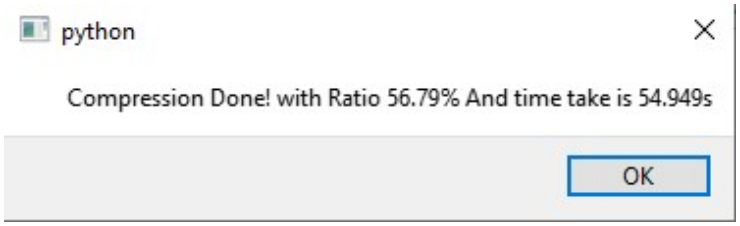


Fig:5.6

Output is displayed in the message box which tells the details about compression ratio and time taken and also status of compression.

Figure 5.7:-Original File

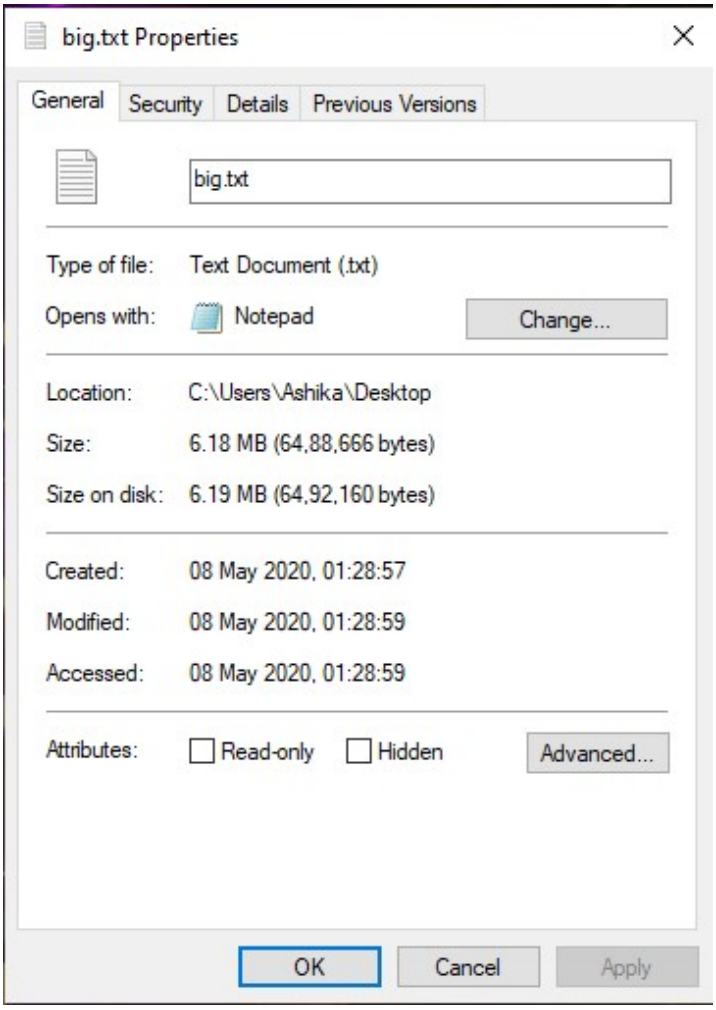


Fig:5.7

In this image we can see the size of the original text file that is 6.18mb

Figure 5.8: Compressed File

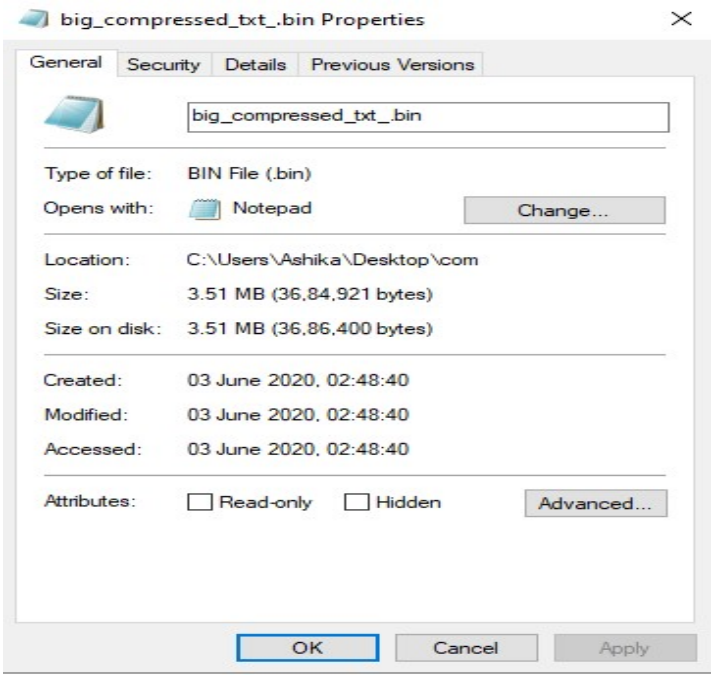


Fig:5.8

In this image we can see the size of the compressed image which is 3.51 half of the size of original file

Figure 5.9:Decompressing the compressed file output

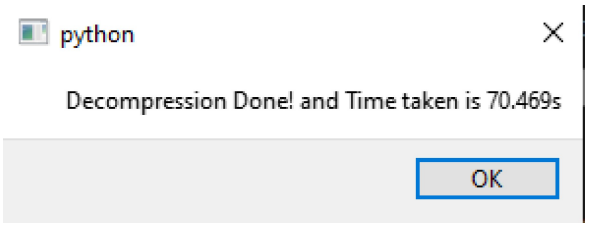


Fig:5.9

Figure 5.10:Multiple text compressor window

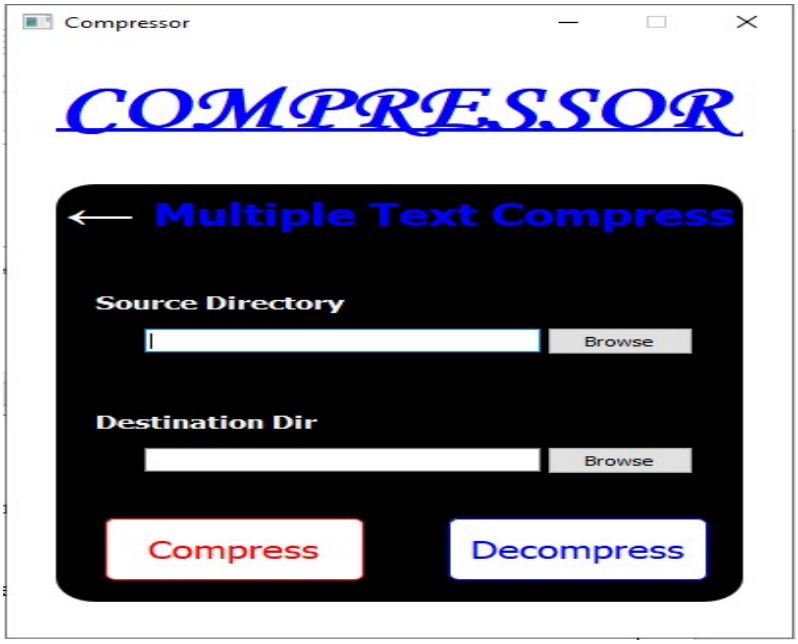


Fig:-5.10

Figure 5.11:-File dialog for source directory

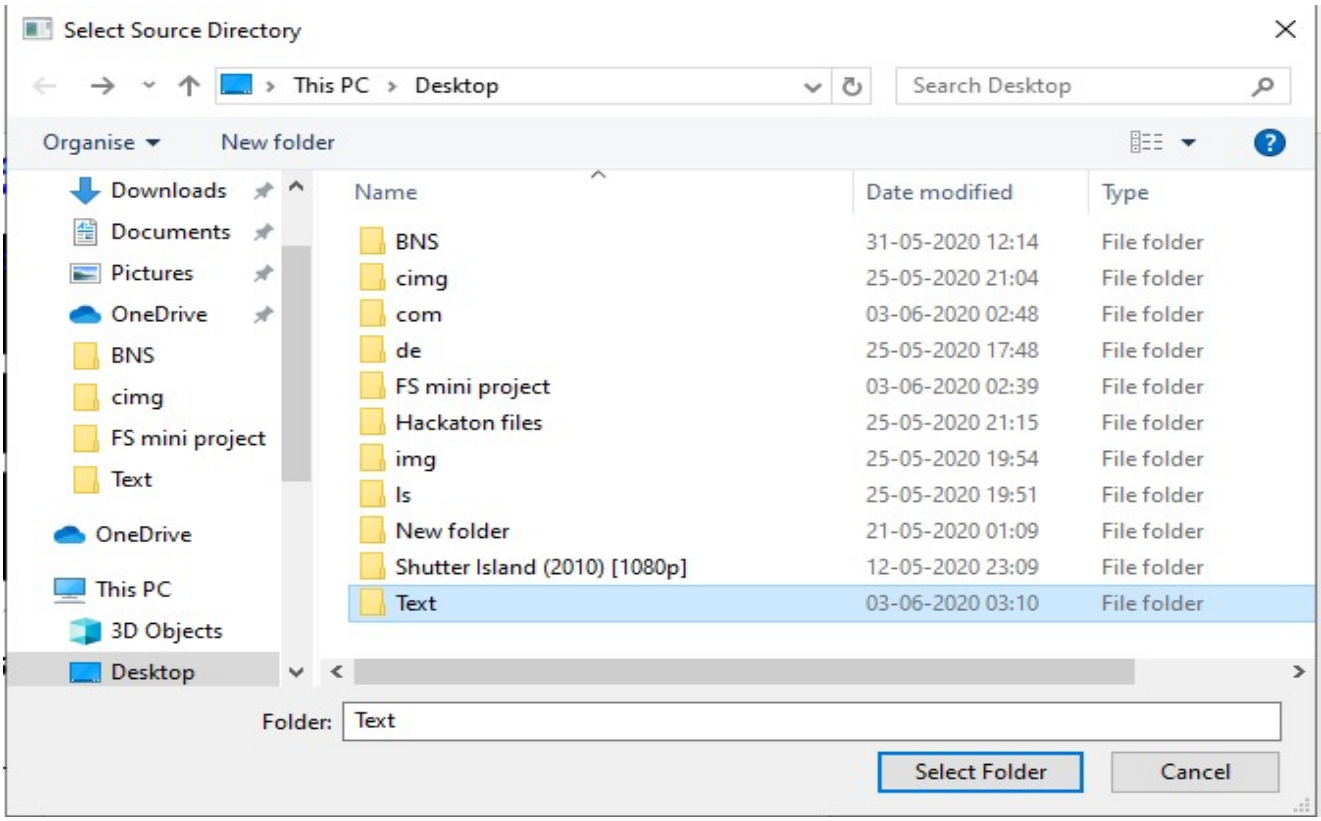


Fig-5.11

Figure 5.12:-File dialog for Destination directory

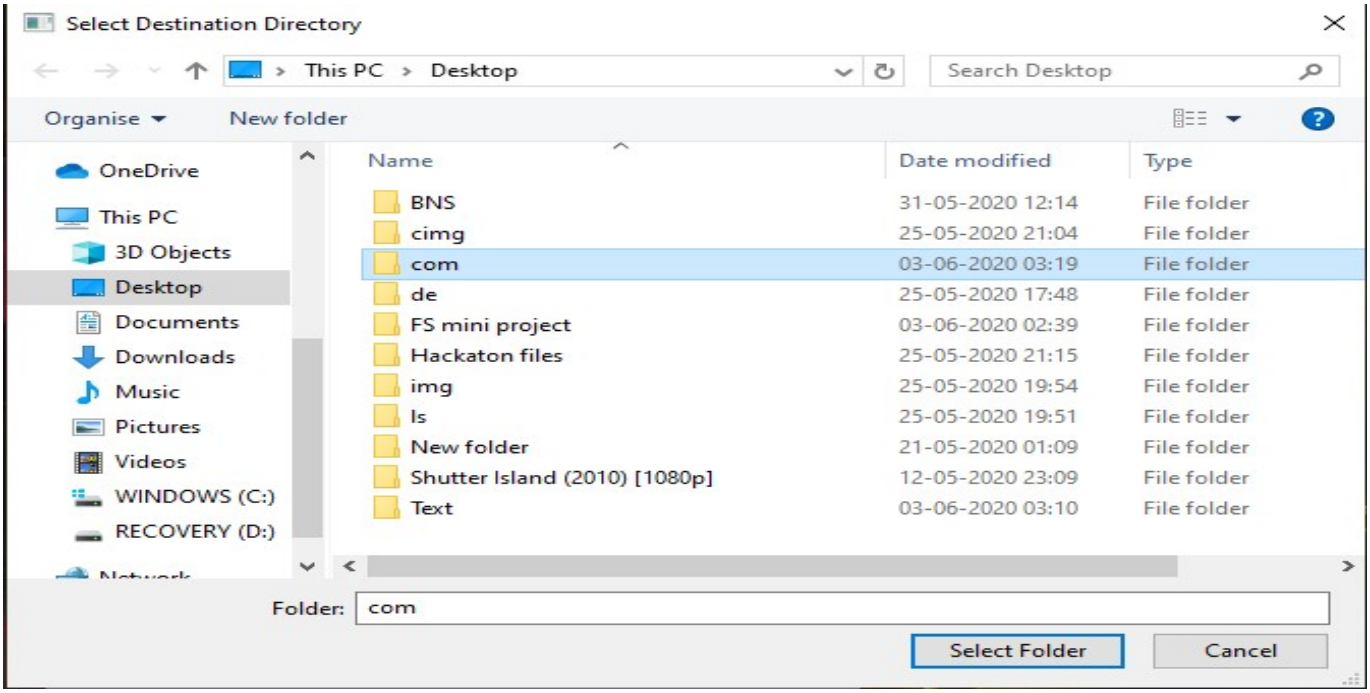


Fig:-5.12

Figure 5.13 :Output

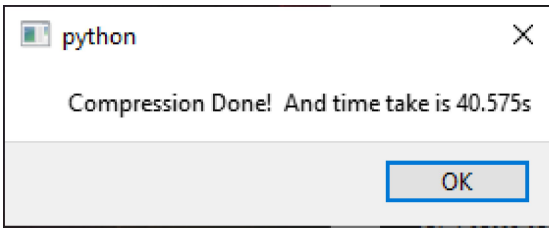


Fig:5.13

Figure 5.14: Original File

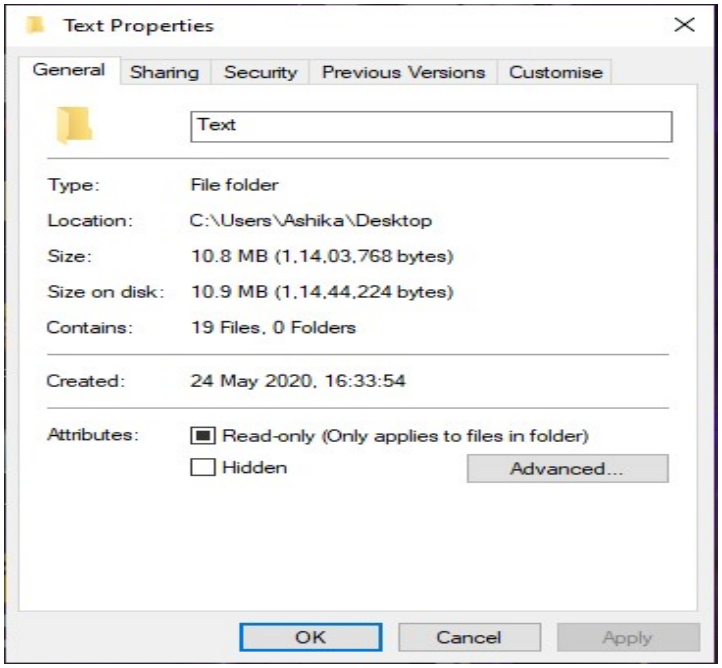


Fig:5.14

In this direcorty it consisted of 10.8mb of data text files

Figure 5.15:Files present in Text directory

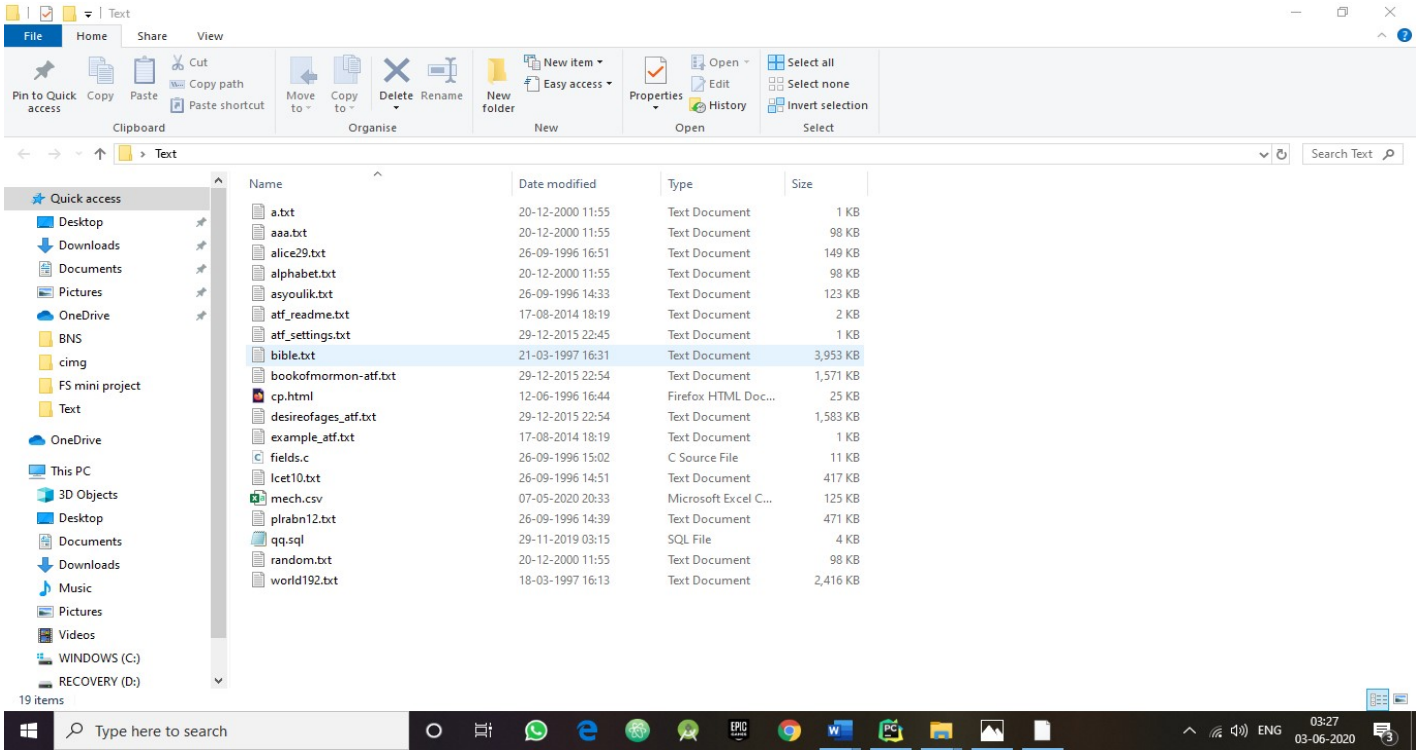


Fig:5.15

Figure 5.16:Compressed Files Directory

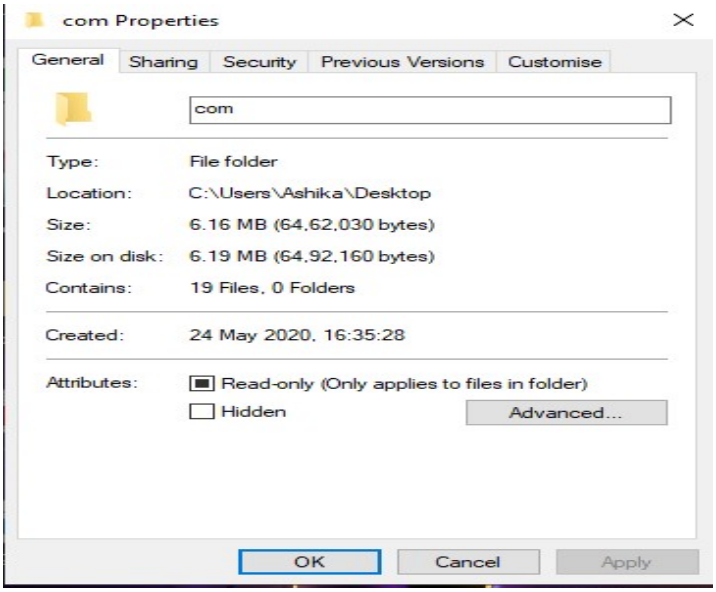


Fig:5.16

As we can see the file size is 6.19 mb which is 61% of the size of original file directory

Figure 5.17:Files present in destination directory

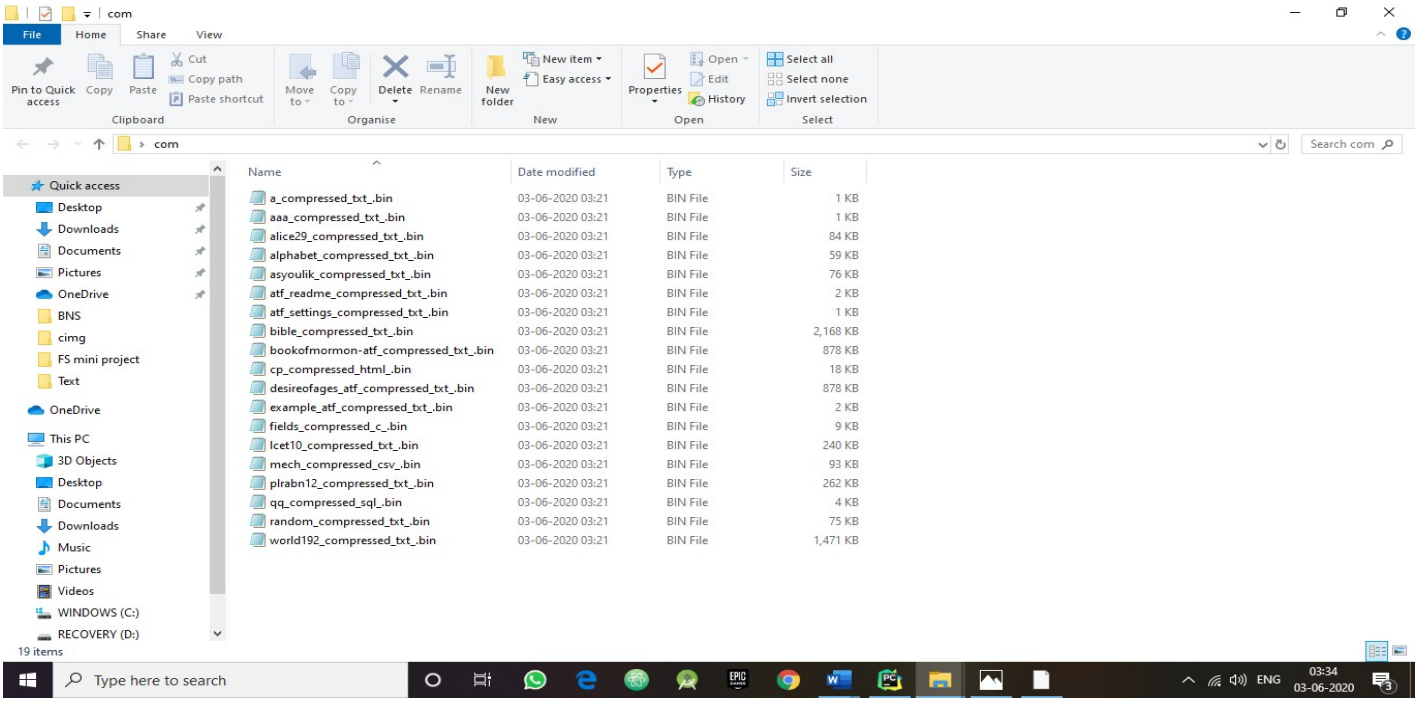


Fig 5.17

From figure 5.15 we can see that there where few other type of files like html, csv, SQL and in fig 5.17 we can see that even those files have been compressed successfully

Figure 5.18:Files dialog for source directory for decompression

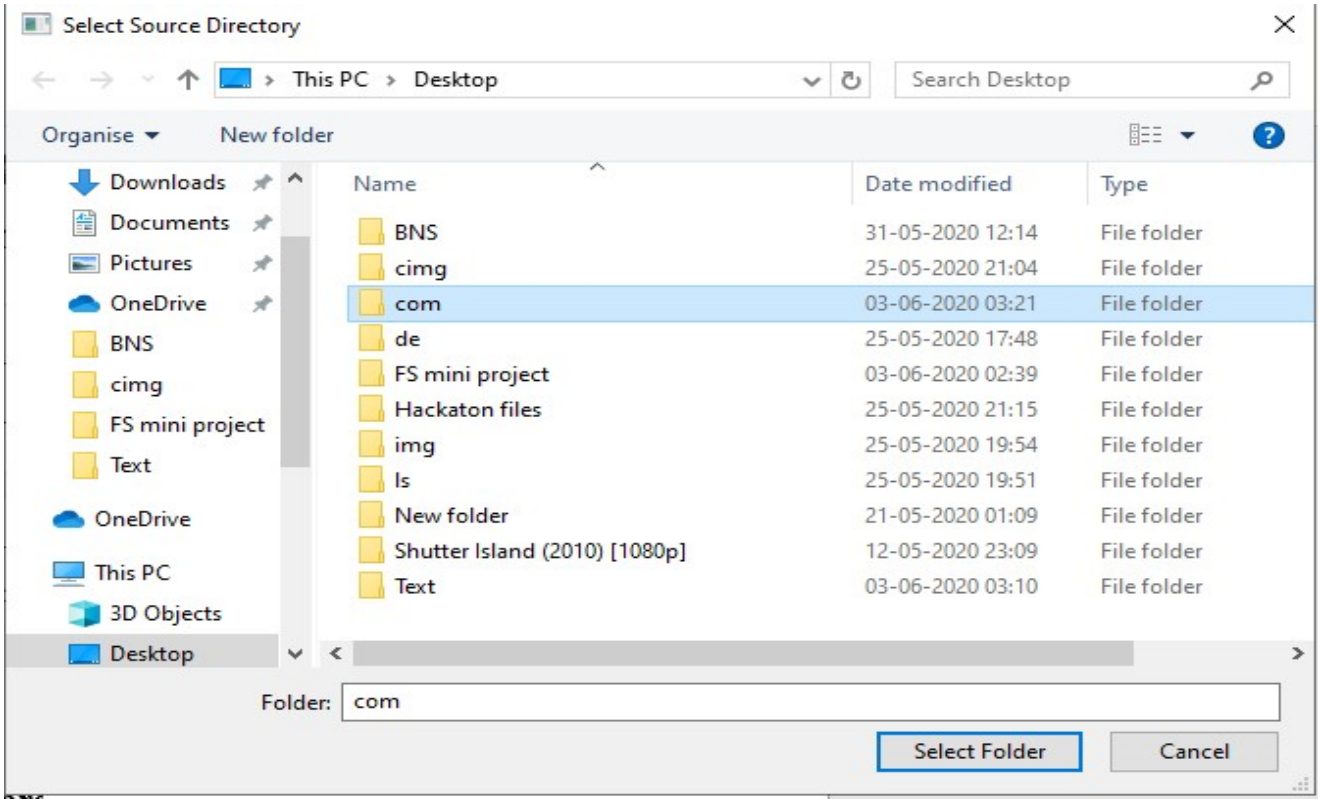


Fig:5.18

Figure 5.19:-File dialog for destination directory for decompression

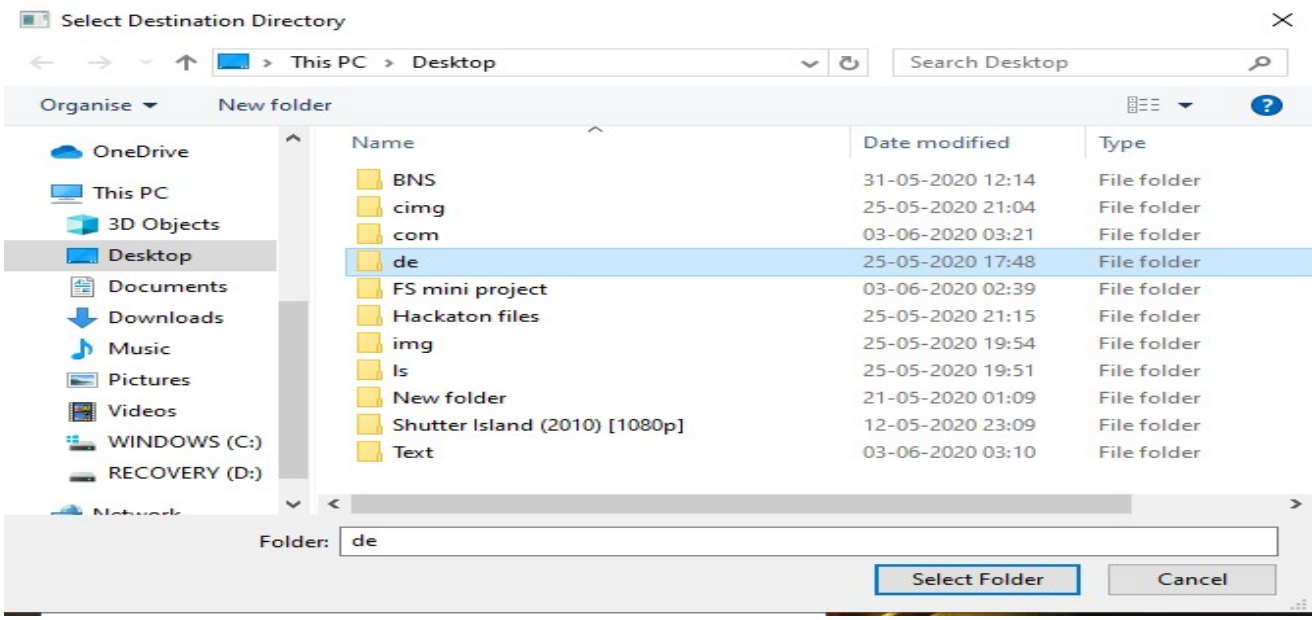


Fig:5.19

Figure 5.20:-Decompression output

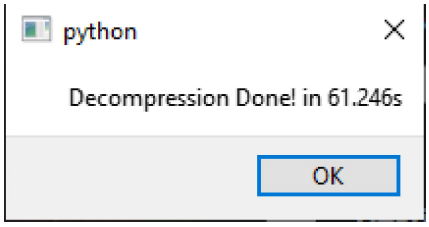


Fig:5.20

Figure 5.21:-Decompression output files

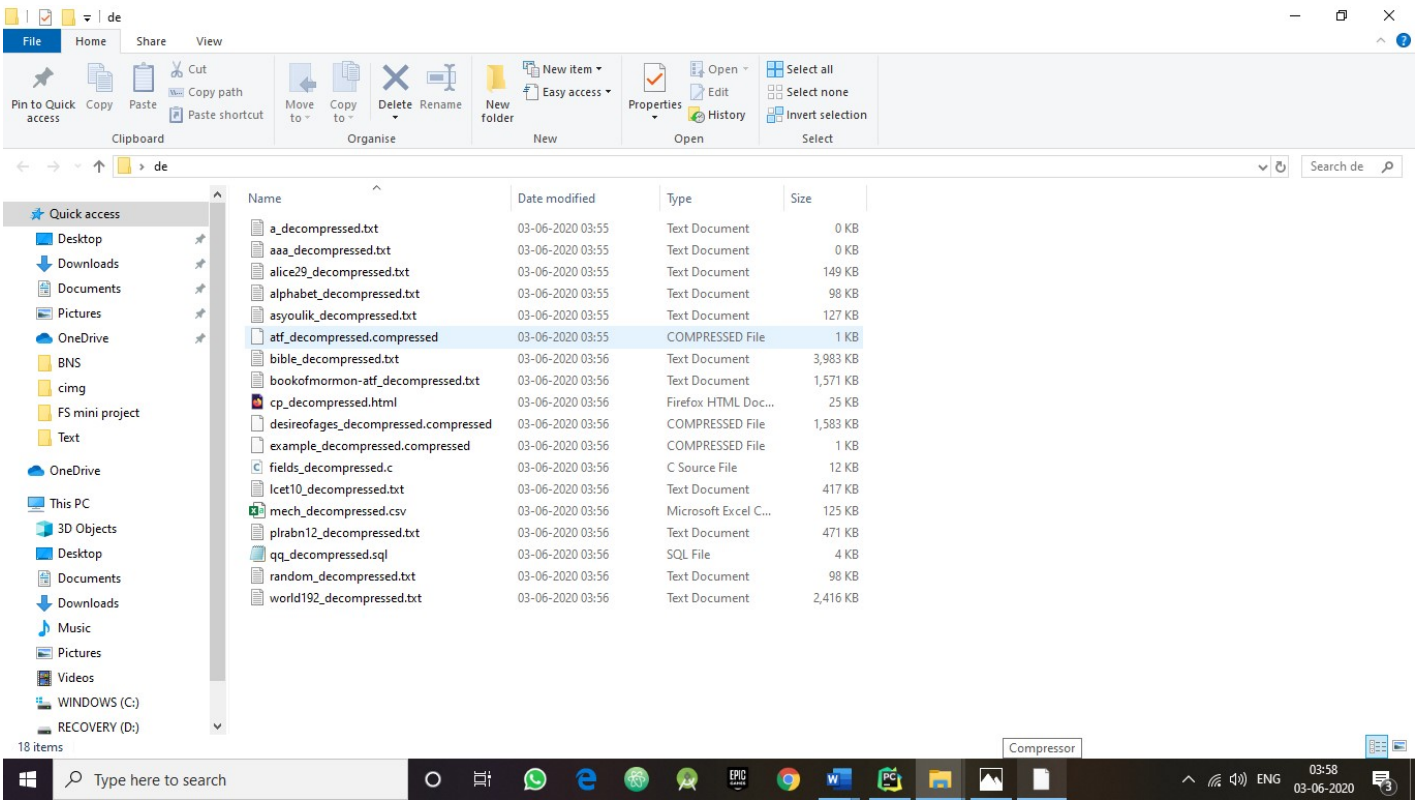


Fig:5.21

In the figure 5.21 we can see files are same as figure 5.15 even the types of the files are restored backed to original

Figure 5.22:-Image option

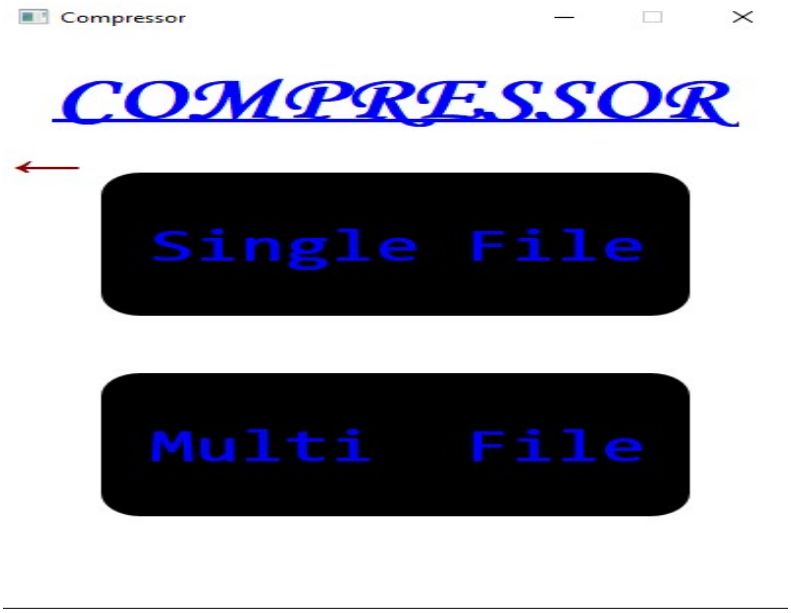


Fig:5.22

Figure 5.23:-Single File image selected option

Fig:5.23

In figure 5.24 it is almost same as single text option but only difference here is quality option in which u can choose high medium low as per user need lower the quality lower is the size.

Figure 5.24:-Choose image file dialog

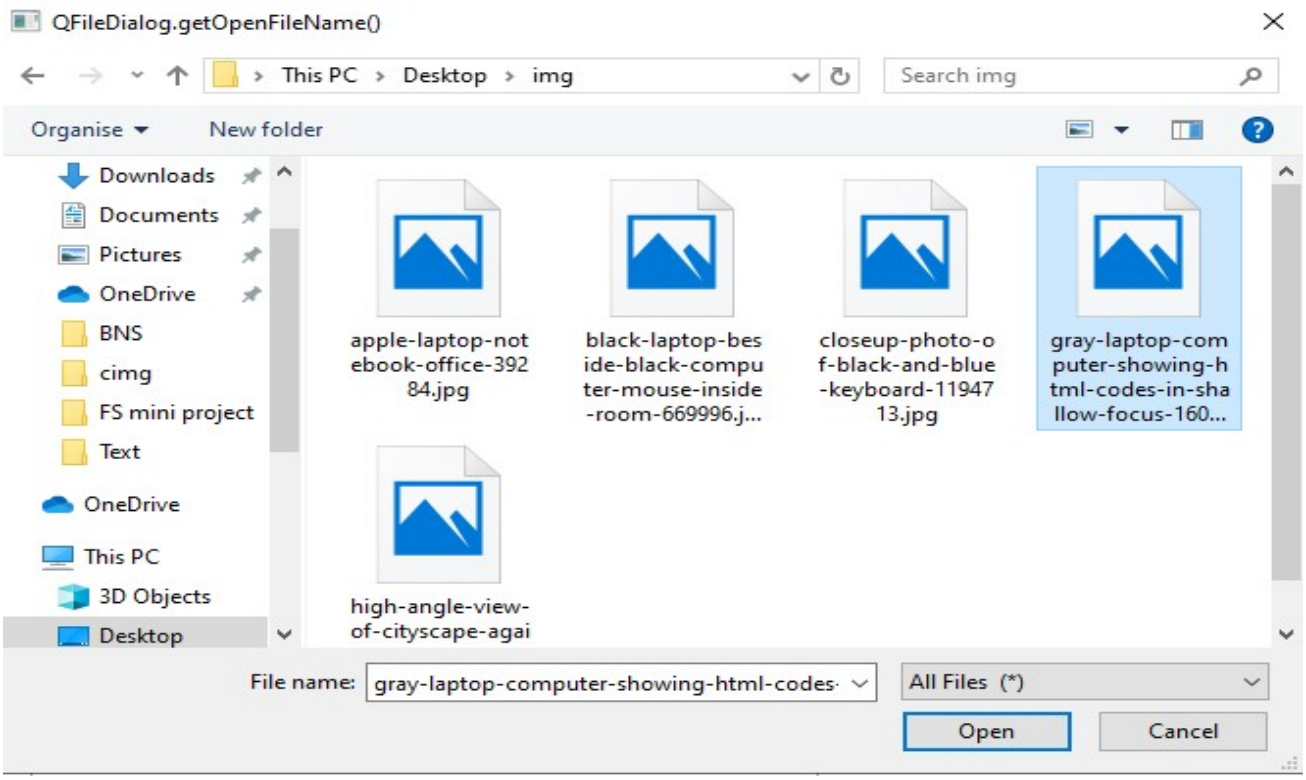


Fig:5.24

Figure 5.25:-Choose directory to save compreesed image file dialog

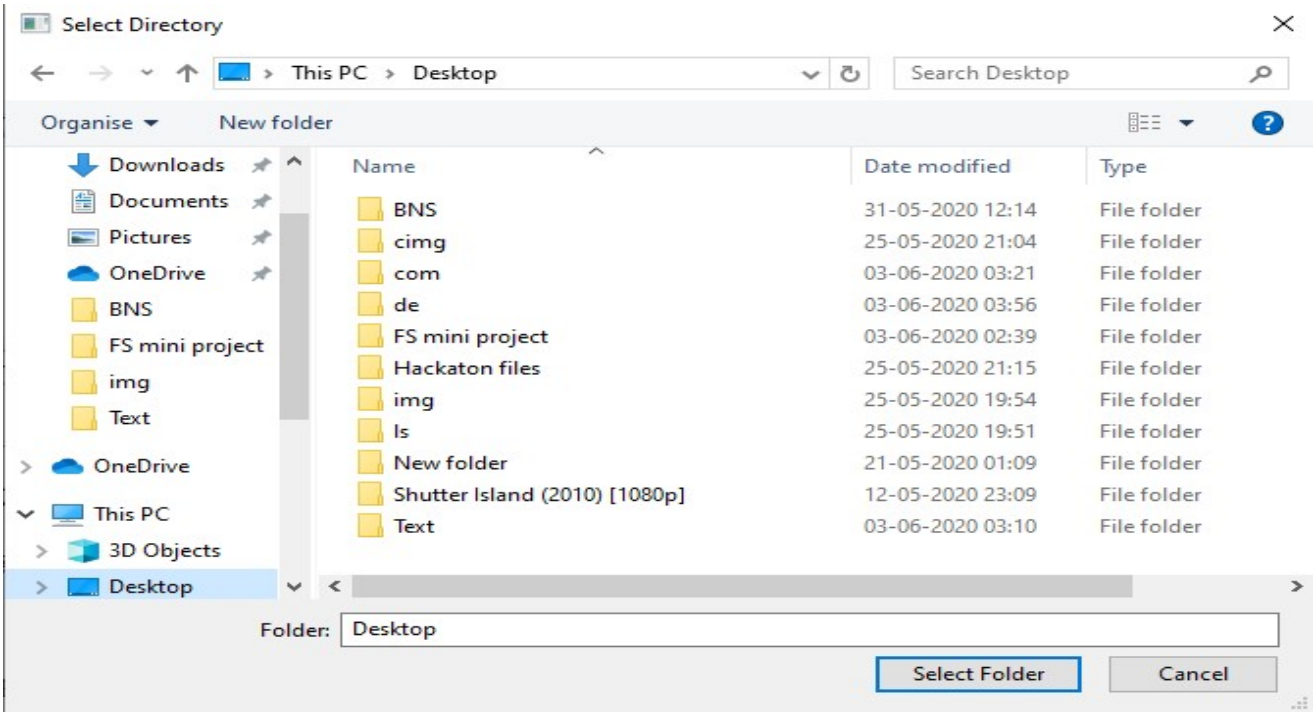


Fig:5.25

Figure 5.26:- Single image compress output

Fig:5.26

We can see that the compression ratio is 0.689% that is because the width of picture is 5000px and I chose to convert it to 500px .as we can see the output image in our message box the image is not so shabby. Also important point is not to give quality more than the size of your display screen if u give also no problem but u wont be able to get ok button as it would have gone out of display u can press enter to exit the message box .Time taken to do it is 0.484s which is super fast.

Figure 5.27:-Properties of original file vs Properties of compressed file

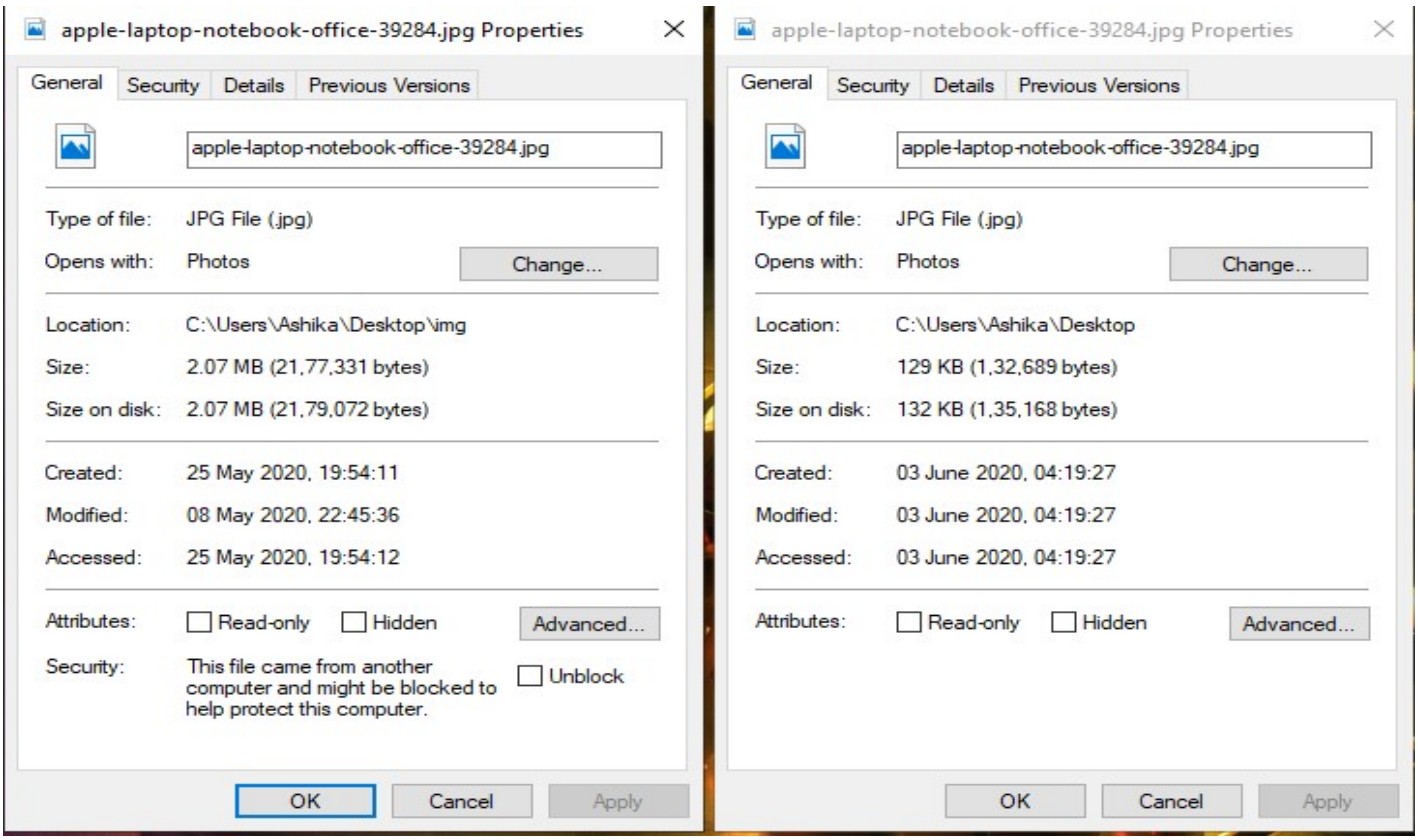


Fig5.27

Figure 5.28:-Multiple Image Compressor Option

Fig:-5.28

Figure 5.29:-Source directory File Dialog for multi image compression

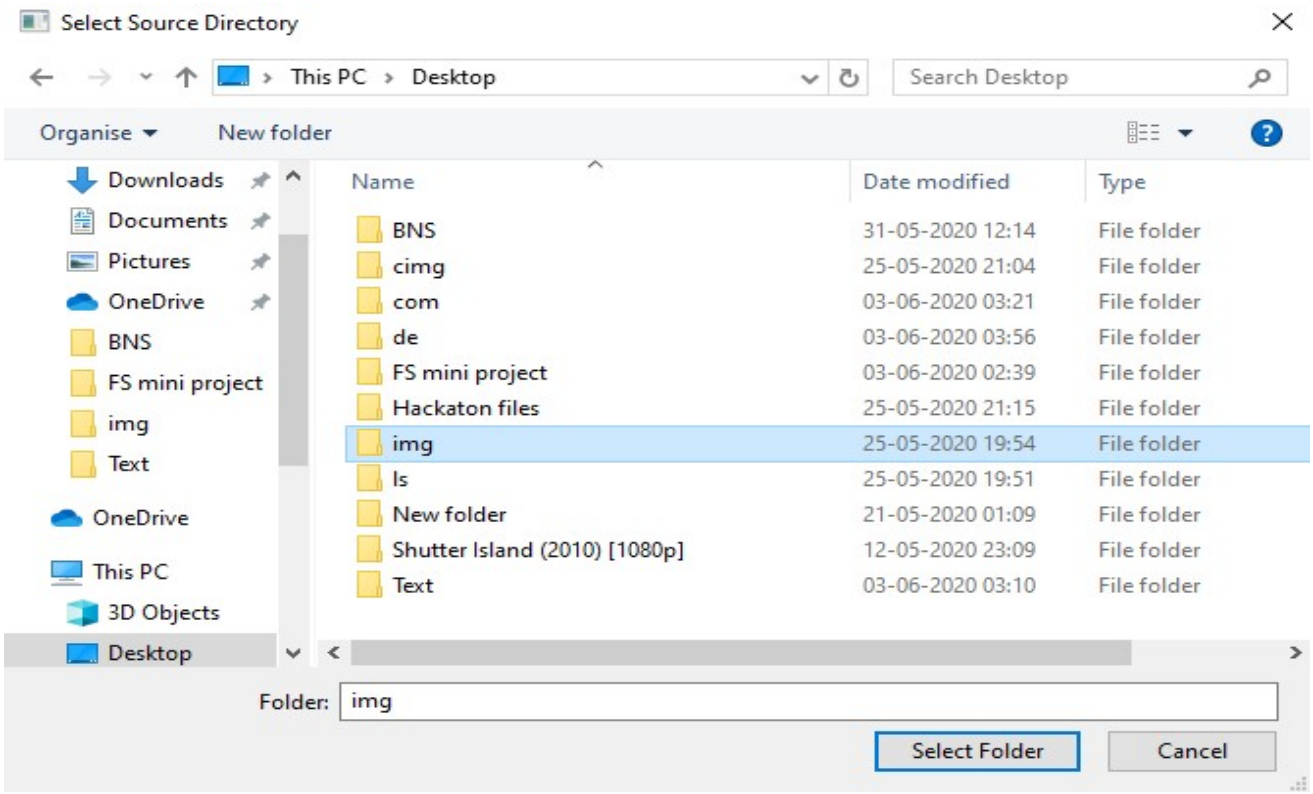


Fig:-5.29

Figure 5.30:-Destination directory File dialog for multi image compression

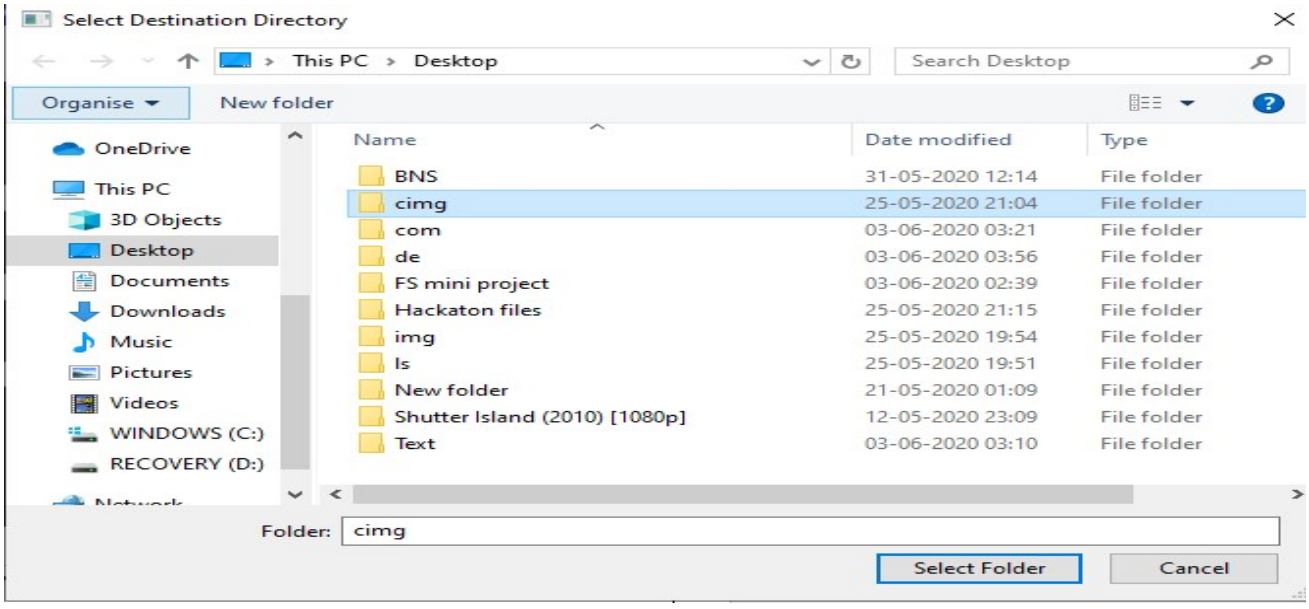


Fig:-5.30

Figure 5.31:-Multiple image compression output

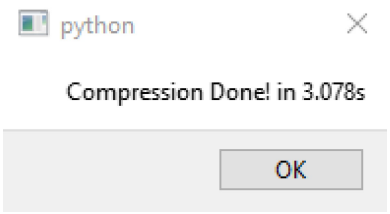


Fig:5.31

Figure 5.32:-Files inside source directory and destination directory

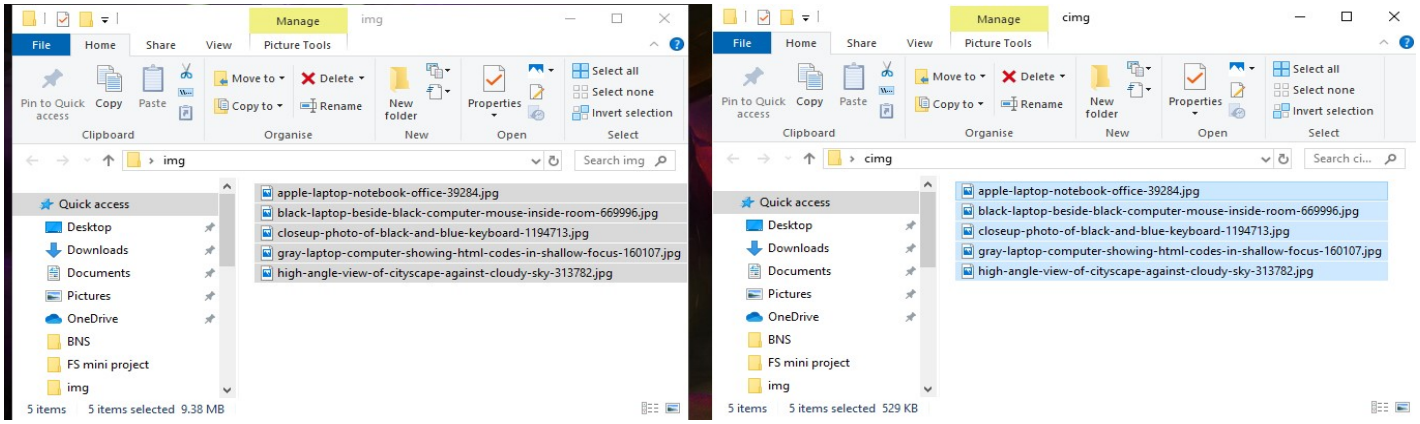


Fig:-5.32

Figure 5.33:-Source directory vs Destination directory multi image compression

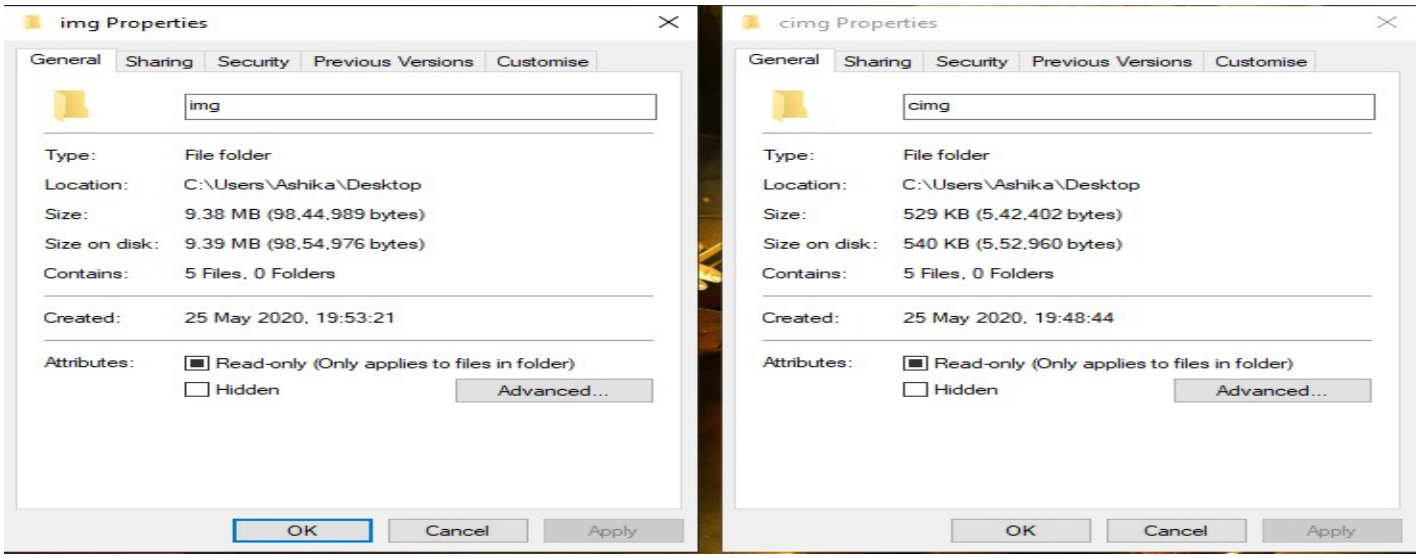


Fig:-5.33

Figure 5.34:-Text Compression Graph(Time V/s Size)

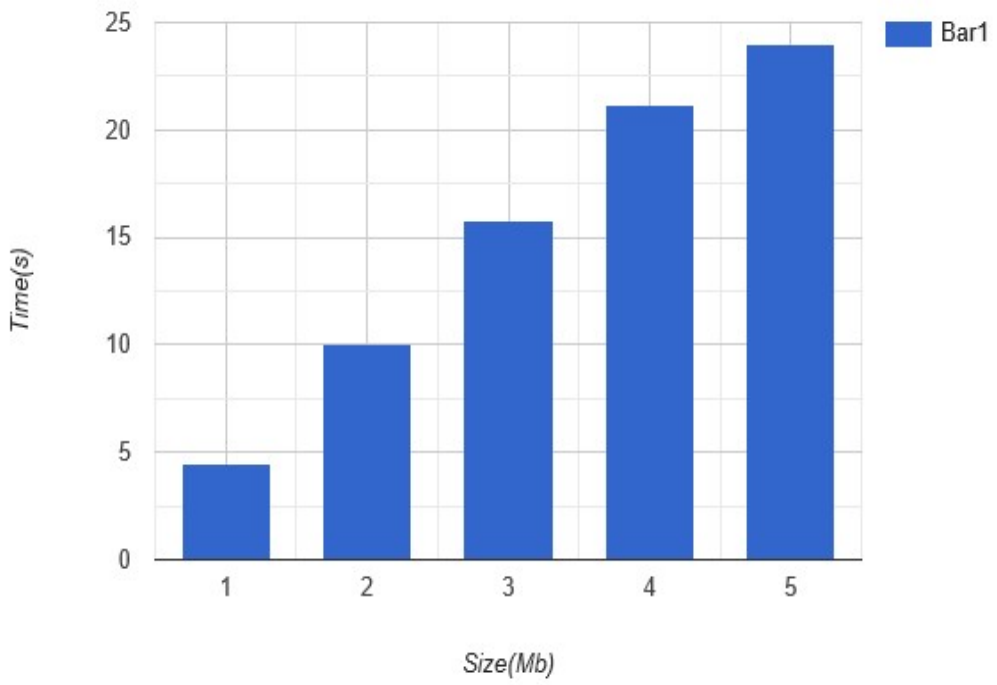


Fig:5.34

Figure 5.35:-Text Compression Graph(Size V/s Compression Ratio)

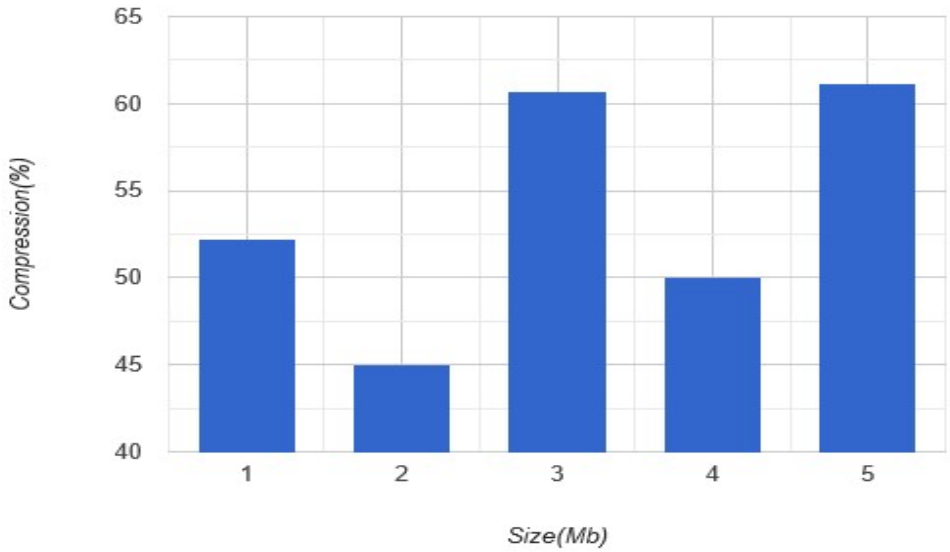


Fig:5.35

Figure 5.36 :- Text Decompression Graph(Time vs Size)

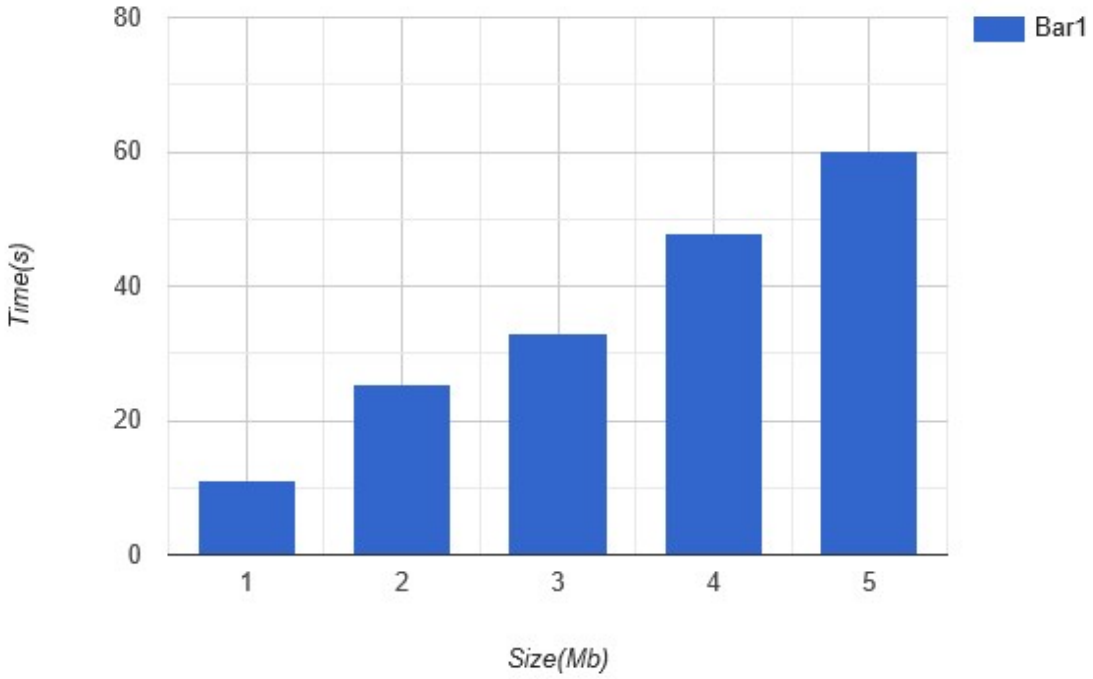


Fig:5.36

Figure 5.37:- Image Compression Graph(Size V/s Compression ratio)

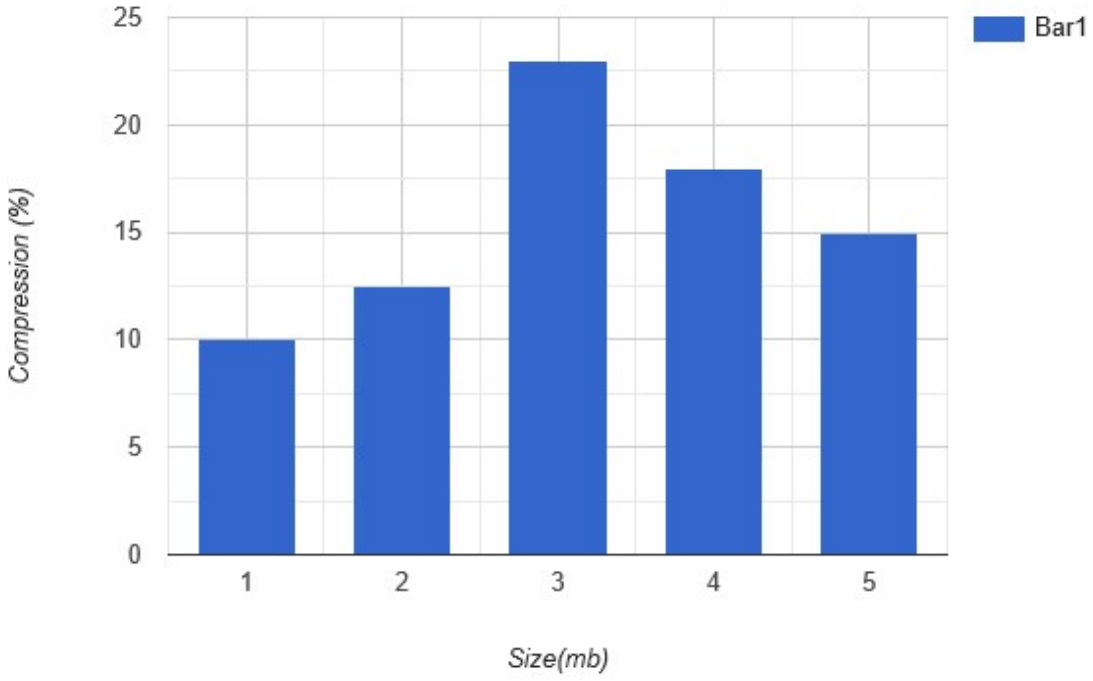
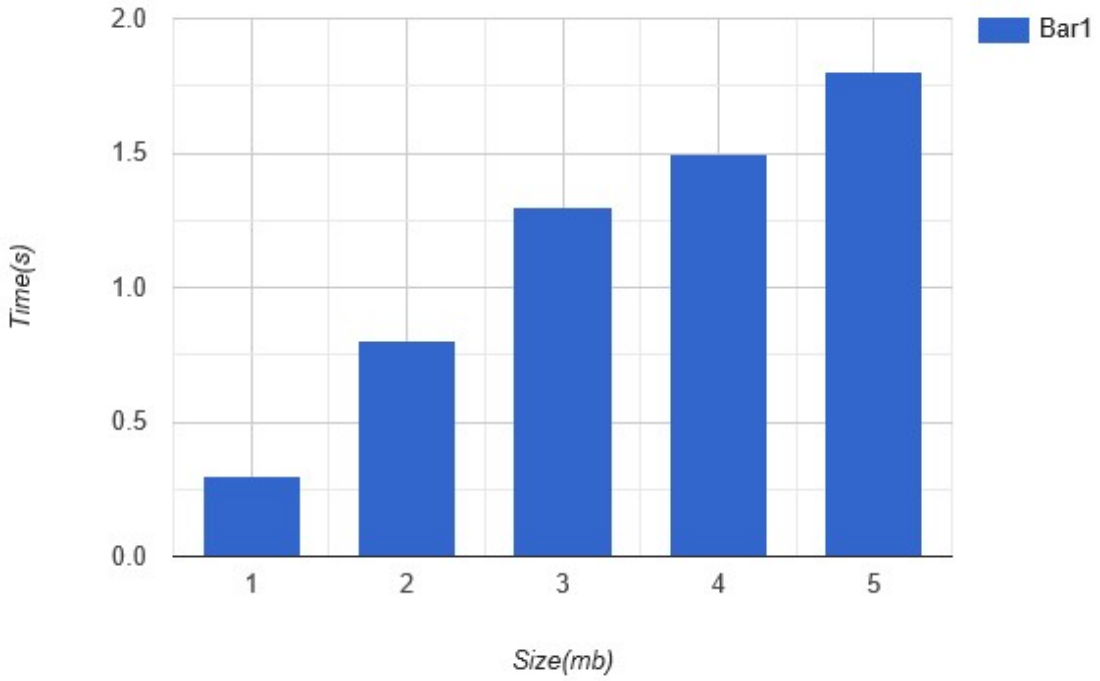


Fig:5.37

Figure 5.38:- Image Compression Graph(Size V/s Time)



CONCLUSION:

In conclusion, data compression is very important in the computing world and it is commonly used by many applications, including the suite of SyncBack programs. In providing a brief overview on how compression works in general it is hoped this article allows users of data compression to weigh the advantages and disadvantages when working with it.

REFERNCES:

1)File Compression Journals:
https://academic.oup.com/comjnl/article-abstract/48/6/677/358346?redirectedFrom=fulltext

2)A Study on Data Compression Using Huffman Coding Algorithms(IJCST) Journal
http://www.ijcstjournal.org/volume-5/issue-1/IJCST-V5I1P10.pdf

3) Book:-File Organization by Binnur Kurt.

4)https://www.tutorialspoint.com/Huffman-Coding-algorithm

5) https://bhrigu.me/blog/2017/01/17/huffman-coding-python-implementation/

6) https://pythonspot.com/pyqt5/

7)https://www.youtube.com/watch?v=8YPjJ93kgTE&list=PLjC8JXsSUrrg2QQgZzRHq_Q1LelPdDjBk