Creating Software To Test Lossless Compression Methods and Algorithms

Quinn Knowles and Paige Schiele

Table Of Contents

Introduction

Data compression is a powerful tool that is used with increasing frequency all over the world. While it may go overlooked it is an essential component that makes the modern digital marketplace possible. When it comes to transmitting data across the internet, almost all large scale file transfers involve some level of compression. This vastly increases the speed of data transfer and helps to get the most out of existing hardware.

Computer hardware is limited and can only store so much data; users of technology may find themselves running out of storage. The solution to this problem is not necessarily expanding the hardware. Compression is a common computer process where files of data are compressed to take up less storage. This process has become essential to making the most out of computer storage. There are two types of compression, lossless and lossy; with lossless, no data is lost during compression. In lossy compression some of the original data is lost, but lossy compression methods still have their own niche. There are many algorithms and techniques used to losslessly compress data. Since there are many different compression methods and many reasons to compress data, it's hard to select what compression method would be the best choice for a given task.

An application to test compression speeds and ratios would be an excellent research tool for determining the efficiency of compression algorithms. This paper intends to present a software solution that helps users test lossless compression algorithms and helps find the method most efficient for their needs.

To limit the scope of this paper the content of the paper and the data collected focuses on data at rest or ‘static’ data. This paper will cover the research into modern compression techniques and algorithms in terms of compression speed, compression ratio, as well as go over results and conclusions of testing the use of multiple algorithms over a narrow scope data set.

The Problem

Running out of storage is a major issue for technology users. This is why laptops, desktops, and some mobile phones have hardware solutions to increase storage such as SD cards, Micro SD cards, or Solid State Drives. However, storage can only be increased so much; once a device has reached its full capacity and there are no hardware solutions available, users are often forced to delete data off their device. Instead of deleting data users could compress files and only decompress them when needed, this would save storage space.

Being efficient with storage space has many benefits. First, if computer users are efficient with their storage it may eliminate the need to buy extra hardware to increase storage which saves money. Second, it increases user experience, no one likes running out of storage; if people are running out of storage often they're going to have a worse user experience.

The nature of lossless data compression, being that it works only by use of tokenizing blocks of repeated data, means that larger files can be compressed to a higher ratio than smaller files. Compressing one folder containing a hundred large files will always yield a higher compression ratio than compressing each of these files individually. However, compressing more and more data together into one file isn't always the answer as it somewhat limits the flexibility of the data, and may even increase the amount of storage used when some files are duplicated.

It is not only static data that can be compressed. As is becoming increasingly common, data in use in a program, as well as data being transmitted over the internet can be compressed. As will be seen with a number of the files that have been tested as a part of this research project, many modern files and file types are being created that already incorporate compression. The reasons for this are not as easily apparent.

When it comes to using data compression in active programs the purpose of doing so has to do with increasingly powerful processors of modern computers. Essentially the bottleneck of modern computers is not processing speed, it is data transfer speed. The concept is that 'processing time is cheap, data transfer time is expensive'.

For example imagine a user trying to open a picture on their desktop. The picture is a massive 100MB. It would take a small amount of time (microseconds) for the user's hard drive to access the file however, the transfer of the data itself from the HDD to the CPU would likely take around a full second. During this time the CPU would be idle, unable to act until the entire file has been received. During this one second period a modern CPU is capable of doing at least tens of millions of instructions. Instead of 100MB if the file was compressed to 60MB, and the CPU were able to work on the data 0.4 seconds faster even if the CPU takes .37 seconds to decompress the file it would be 3% faster, potentially providing time for hundreds of thousands of instructions and freeing up the connection between the CPU and HDD for other processes that need to access data.

Ultimately what the concept displayed here is that modern CPUs are calculating faster than the motherboard can feed them data to calculate, as can be seen by the existence of concepts such as "Speculative Execution". If instead of letting the CPU sit idle it were busy decompressing data sent to it, the motherboard would be able to send the CPU more data speeding up the machines effective data transfer rate. This can be seen in files like .MKV and .JPEG which have some built-in compression.

The other advantage compression can give data in transit is data transfer over the world wide web. Much like the earlier example, data transfer speed is at a premium. Internet service providers in the United States make up a $138.7bn industry, an industry of many many people across the country competing to use the same data conduits. If files are compressed before being transferred, they take up less space and less time on the data conduits that connect the world. Greatly reducing the time it takes to send data from one place to another, and opening up room for other users.

The problem this research paper addresses is one of static data, not data in transit, but it's important to understand the prevalence of data compression in all of its facets. Data compression is everywhere.

Metrics

Each algorithm has been compared based on compression speed, and compression ratio. The faster it is to compress or decompress the data, the more accessible the compressed data is. Compression ratio will be taken into account; compression ratio is the size of the compressed file divided by the initial size. The point of compression is to turn larger files into smaller ones to save space so the smaller the file is compared to the decompressed size, the better the compression algorithm is. With this being said, the lowest compression ratio might not always be the best option for users. The algorithm chosen for compression will ultimately depend on the needs of the user. If the user needs the files compressed quickly and compression ratio matters less a faster algorithm should be chosen. Each situation is unique which is why there is no single compression method that is used.

An example of how compression speed is important can be seen among a number of the file types used during this research project. As was discussed in the previous section, .MKV and .JPEG files incorporate some levels of compression in their static state. Both of these file types exist in what is a “usable” state. As in the files are meant to be accessed, viewed, and interacted with by the user immediately. If the file were so compressed that it delayed the user from accessing it, rather than speed up the process, then the compression being used is not fulfilling its intended role.

Likewise for data storage, the speed at which data is compressed is important to the user. In this interest, and to account for the importance of this statistic the speed at which each file was compressed was recorded, and the average speed per byte was found for each algorithm over the given data set.

Other metrics that were recorded or calculated include compression ratio variance, compression ratio standard deviation, and average compression ratio over the given dataset.

Techniques

In data compression there are two main types of compression techniques separated into their own categories. Lossless and Lossy compression techniques are the two broad categories in which all compression types can be contained. When a file is compressed it is taken from a state where it takes up more bytes of data to a state where it takes up less bytes of data. This can be achieved either by omitting some of the file's data, or by representing the data in a more concise way.

When a file is compressed using a lossless data compression algorithm all of the data the file contained is still present. If a file has been compressed using lossless compression, then upon decompressing the file it will be identical to how it had been before being compressed. This paper has a focus on lossless compression, however it is important to establish what Lossy compression is in order to fully understand lossless compression.

Lossy compression is where in order to shrink the size of the file, some concessions have to be made on data integrity. Lossy compression can be seen in things such as video streaming. When streaming a video often a user may be able to select multiple levels of quality, often any range from 1080p to 144p. The file exists, somewhere, in its most data rich form (the highest quality version), but that might not fulfill the needs of the user based on the rate of data transfer they have available to them. A user is therefore able to request lower quality versions of the original data, in order to accommodate their internet speed. This is accomplished using lossy data compression, which is capable of high 'compression ratios'. This comes at the cost of data integrity, but in some cases that is acceptable.

Many lossless algorithms use similar techniques to compress data. Huffman coding, run-length encoding, prediction by partial matching, and LZ77. Huffman coding is a way to compress data originally developed by David Huffman. It uses a tree to reduce bytes used. The way Huffman coding works is by reducing bytes based on frequency. The more a character is used the less bytes will represent it. Run length encoding is good if there are long stretches of the same data; run length encoding works by storing a value/character and then how many times that value/character is repeated. If the data is too different and there is not a lot of similar data in a row, run length encoding will likely increase the file size rather than compress it. Prediction by partial matching, put simply, uses the last chunk of data to predict the next. Certain file types also already implement lossless compression methods to keep the files small. PNG, BMP, and GIF are all examples of file types that do this. These files however can be compressed down even further using certain tools and applications.

Data compression is the process of reducing the total number of bytes a file uses in storage. Data compression is important and extremely useful because computers can only store so much data; compression helps make the most out of the computer storage available. There are many types of compression, all falling into two broad categories: Lossy Data Compression, and Lossless Data Compression. There are also different compression methods used depending on the type of file being compressed.

Lossless Compression Algorithms

There are many lossless compression algorithms to choose from, for this project the algorithms that will be compared are, PPMd, 7zip: Deflate, 7zip: LZMA2, Windows Compress. Deflate is a compression algorithm that uses Huffman encoding and concepts learned from LZ77 to compress data. LZMA2 uses dictionary compression to compress files and is an upgrade from the LZMA algorithm. PPMd uses prediction by partial matching to compress data. There are also applications that are used for file compression such as WinRAR and Adobe Acrobat. There are lots of free applications online that users can use to compress files. The software solution created focuses on comparing ZIP compression methods.

In the category of lossless data compression there are many different algorithms of varying sophistication and complexity, each of them utilize some combination of the following tokenization techniques:

- Run-length encoding - compression of repeating sections of data into a single data segment with an indicator of how many times the section needs to be repeated.

- Dictionary Coder - Similar to run length encoding, instead using pointers to replace data segments.

- Prediction by Partial Matching: Involving the use of context modeling. Prediction by partial matching uses symbol ranking and arithmetic functions.

- Deflate - Data segments are broken into blocks of *n* length, with the addition of a 3-bit header.

- Context Mixing - In which the next symbol predictions of two or more statistical models are combined to yield a more accurate result.

- Huffman Encoding - Involving a binary tree, sorted by frequency. Somewhat dated because of its linear scalability.

All of these techniques share one thing in common, which is the core tenet of lossless data compression. Each of those techniques listed above rely on tokenization of data in order to save space. It should be noted that when it comes to lossless compression, there can be no guarantee that a particular file can be compressed at all. Sometimes a file contains so little duplicate blocks that the algorithm is unable to make up for the space that the file header adds to the file.

Design Summary

Before any programming was started the team first researched compression methods and techniques and decided to focus on lossless compression. After research was mostly completed, lossless compression algorithms were chosen. Planning began on how to compare the data and what tests should be run to best compare the algorithms.

After research was complete a Python 3 program was created to compare the algorithms and techniques. Many different file types were tested including, PDF, PNG, MP3, WAV, MP4, MKV, JPEG, and DOC. There was a large range of file sizes tested. The program works by using functions from the time, os, pathlib, and subprocess libraries. To get the program to run correctly on a computer, Python 3.10 must be installed and 7-zip. Python 3.10 and 7-zip paths must also be added to the environment variable "path" under system variables.

The program itself can be run from the file in whatever folder it is in or by using the command prompt. In order to work properly the user should place the script in the root directory of the files they want to compress. After that the program can be started, entering "py "compress drive" or by running the program from file explorer. Then the program will cycle through all files in the folder the program is stored in and compress every one using the Deflate, LZMA2, Windows Compress, and PPMd methods.

The user is given a couple different options using simple command prompt input. First the user will select whether they want to test the files or compress the files. Selecting 'Test' will cause the program to recurse through all individual files and subdirectories in the root directory. The test function creates a data file on the C:\ drive at C:\Capstonedata\data.txt . The following data is recorded for every compression method on every file: Original size, compressed size, compression ratio, and time taken to compress. Afterward, the compressed files are all removed and the directory is returned to its original state.

The more practical application of the script is the 'C' or compress command. After this command is entered the user will be prompted again with a choice between compressing all individual files, and compressing only the files and folders in the root directory. As with the earlier option all objects are run through four different compression algorithms. In this mode only three of the compressed files are deleted, while the compressed file with the best compression ratio remains. In this way, the script can be used as an aid to compress files or folders on a drive, saving space for the user.

The program was run on 16 different files. There were 2 files selected for each of the file types listed above; one was a smaller file and one was larger. This was to see how the compression methods work on various sizes of files. The files were also selected so that they would be similar in size to another similar file type. For example, a 100 kb and 1000 kb file was tested for both PDF and .doc. This was to show the differences in compression ratio and speed between the file types.

Program Code

The following code is the current version of the program. This version is written in Python 3.10. It currently compresses data using the methods PPMd, Deflate, LZMA2, and Windows Compress. The results, including compression times and ratios, are written in a text file. Comments have been added to explain what each part of the code does.

```
#this script has the following dependencies: Windows, 7zip, and
Python 3.10
#both 7zip and Python must have their path listed in windows
environment variables
import os
import time
import subprocess
import pathlib
#input subdir, file, array where subdir is the path from root,
file is the file name, and array is an array of integers
#compresstest() runs the given file through 4 compression
algorithms LZMA2, Deflate, PPMd, and windows' native compression
#each algorithm creates and stores its own version of the file
and stores it in its respective compressed format
#The files size will be recorded and the files deleted
#return: array of integers
def compresstest(subdir,file,array):
    t= ';'
    FilePath = os.path.join(subdir, file)
#create full file path for reference
    OGFileSize= os.path.getsize(FilePath)
#obtain original file size to reference for compression ratio
################################################################
#####################################
#begin mx test
    array[0] += int(OGFileSize/1000)            #add file size
to data set to be returned as output
    if(OGFileSize==0):                          #scrub cases
where the file is a stub
        return
    RealTime = time.time()
#Take start time for Data collection
    MXnamestart=file+'mx9test.7z'
#setup file name for mx9 test
    MXname = os.path.join(subdir,MXnamestart)
#Create a file disignation for mx9 compressed file
    cmd = ['7z', 'a', MXname , FilePath, '-mx9']
    subprocess.run(cmd)                              #Run mx9 command
    CompMethod = '7zip: LZMA2 '
    EndRealTime = time.time()                        #End Timer
    ENDFileSize = os.path.getsize(MXname)
#get new file size (in bytes)
    MXS= ENDFileSize
    MXdata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
    array[1] += int(ENDFileSize/1000)
#add file size to data set to be returned as output
#end mx test
################################################################
######################################
#begin deflate test
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
 #Take start time for Data collection
    Deflatenamestart=file+'Deflatetest.7z'
#setup file name for deflate test
    Deflatename = os.path.join(subdir,Deflatenamestart)
    cmd = ['7z', 'a', Deflatename , FilePath, '-m1=deflate']
    subprocess.run(cmd)
#Run Deflate command
    CompMethod = '7zip: Deflate '
    EndRealTime = time.time()                        #End Timer
    ENDFileSize = os.path.getsize(Deflatename)
#get new file size (in bytes)
    DeflateS = ENDFileSize
    Deflatedata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
    array[2] += int(ENDFileSize/1000)
#add file size to data set to be returned as output
#end deflate test (DeflateS)
################################################################
######################################
#begin Windows compress-archive test
    W=0
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
#Take start time for Data collection
    Winfile = '\"' +os.path.join(subdir,file)+'"'
#setup file name for win test
    Winname = FilePath+ 'wint.zip'
    pwrname = '\"' + Winname +'\"'
    cmd = ['powershell', 'compress-archive', Winfile ,  pwrname
]
    subprocess.run(cmd)
#Powershell compress-archive -Path String -DestinationPath
String
    Winname = os.path.join(subdir, Winname)
    CompMethod = 'Windows Compress '
    EndRealTime = time.time()                           #End Timer
    if os.path.isfile(Winname):                         #windows
compress archive fails when used through powershell on large
files. This is for error catching.
        ENDFileSize = os.path.getsize(Winname)
#get new file size (in bytes)
        W=1
        WinS = ENDFileSize
        array[3] += int(WinS/1000)
#add file size to data set to be returned as output
        Windata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
    else:
        Windata = str(CompMethod) + t + 'FAILURE'
#record errors
#end Windows compress-archive test
#################################################################
######################################
#begin PPMd
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
 #Take start time for Data collection
    PPMdnamestart=file+'PPMdtest.7z'
#setup file name for PPMd test
    PPMdname = os.path.join(subdir,PPMdnamestart)
    cmd = ['7z', 'a', PPMdname , FilePath, '-m1=PPMd']
    subprocess.run(cmd)
#Run Deflate command
    CompMethod = '7zip: PPMd '
    EndRealTime = time.time()                          #End Timer
    ENDFileSize = os.path.getsize(PPMdname)
#get new file size (in bytes)
    PPMdS = ENDFileSize
    PPMddata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
    array[4] += int(ENDFileSize/1000)
#add file size to data set to be returned as output
#End PPMd
#################################################################
######################################
#Record which file had the best compression ratio:
    if(W==1):
        if(MXS <= DeflateS and MXS <=WinS and MXS<=PPMdS):
            array[5] +=int(MXS/1000)
        elif(DeflateS <=MXS and DeflateS <=WinS and DeflateS<=
PPMdS):
            array[5] += int(DeflateS/1000)
        elif(WinS<=MXS and WinS <=DeflateS and WinS <=PPMdS):
            array[5] +=int(WinS/1000)
        else:
            array[5] += int(PPMdS/1000)
    elif(W==0):
        if(MXS<=DeflateS and MXS<=PPMdS):
            array[5] +=int(MXS/1000)
        elif(DeflateS<=MXS and DeflateS<= PPMdS):
            array[5] +=int(DeflateS/1000)
        else:
            array[5] += int(PPMdS/1000)
#end record
################################################################
######################################
#clear compressed versions of files
    os.remove(MXname)
    os.remove(Deflatename)
    if os.path.isfile(Winname):
        os.remove(Winname)
    os.remove(PPMdname)
#compressed files removed
################################################################
######################################
#write results to data file
    f= open('C:\Capstonedata\data.txt', 'a')
    f.write('\n')
    f.write(MXdata)
    f.write('\n')
    f.write(Deflatedata)
    f.write('\n')
    f.write(Windata)
    f.write('\n')
    f.write(PPMddata)
    f.close()
```

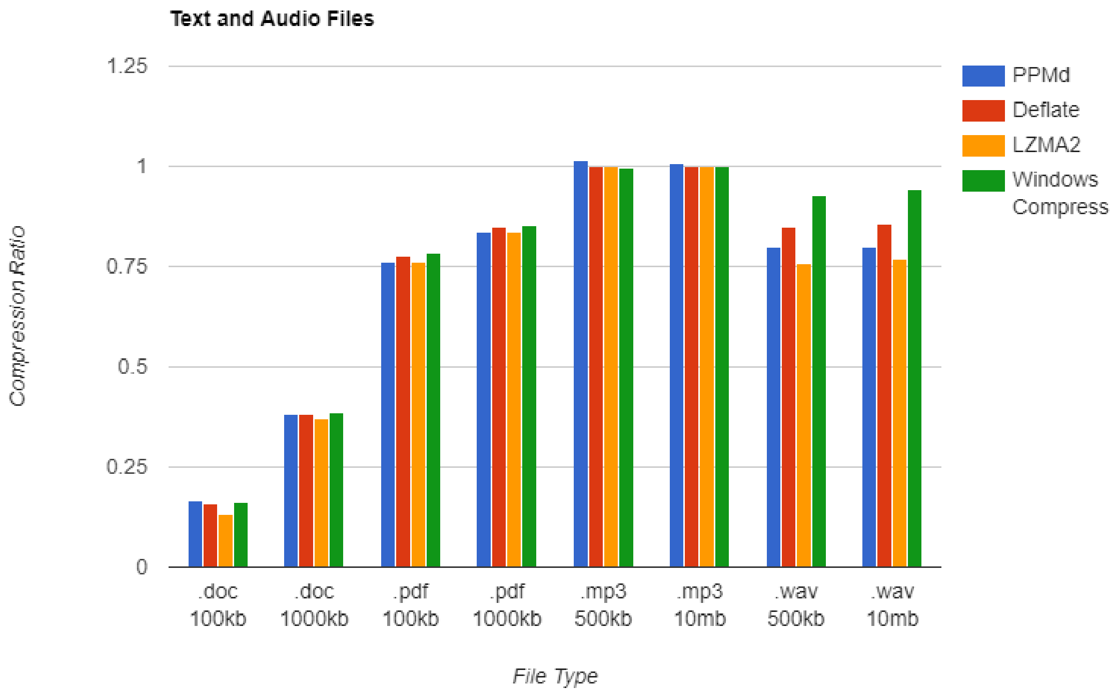
```
#Write complete
    return(array) #end of compressiontest function

#input subdir, file where subdir is the path from root, file is
the file name
#compress() runs the given file through 4 compression algorithms
LZMA2, Deflate, PPMd, and windows' native compression
#each algorithm creates and stores its own version of the file
and stores it in its respective compressed format
#The files size will be recorded and the all but the smallest
file will be deleted
#return:
def compress(subdir,file):
    t= ';'
    FilePath = os.path.join(subdir, file)
#create full file path for reference
    OGFileSize= os.path.getsize(FilePath)
#obtain original file size to reference for compression ratio
################################################################
#####################################
#begin mx test
    if(OGFileSize==0):
#scrub cases where the file is a stub
        return
    RealTime = time.time()
#Take start time for Data collection
    MXnamestart=file+'mx9test.7z'
#setup file name for mx9 test
    MXname = os.path.join(subdir,MXnamestart)
#Create a file disignation for mx9 compressed file
    cmd = ['7z', 'a', MXname , FilePath, '-mx9']
    subprocess.run(cmd)                             #Run mx9 command
    CompMethod = '7zip: LZMA2 '
    EndRealTime = time.time()                       #End Timer
    ENDFileSize = os.path.getsize(MXname)
#get new file size (in bytes)
    MXS= ENDFileSize
    MXdata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
#end mx test
################################################################
######################################
#begin deflate test
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
#Take start time for Data collection
    Deflatenamestart=file+'Deflatetest.7z'
#setup file name for deflate test
    Deflatename = os.path.join(subdir,Deflatenamestart)
    cmd = ['7z', 'a', Deflatename , FilePath, '-m1=deflate']
    subprocess.run(cmd)
#Run Deflate command
    CompMethod = '7zip: Deflate '
    EndRealTime = time.time()                    #End Timer
    ENDFileSize = os.path.getsize(Deflatename)
#get new file size (in bytes)
    DeflateS = ENDFileSize
    Deflatedata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
#end deflate test (DeflateS)
################################################################
######################################
#begin Windows compress-archive test
    W=0
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
#Take start time for Data collection
    Winfile = '\"' +os.path.join(subdir,file)+'"'
#setup file name for win test
    Winname = FilePath+ 'wint.zip'
    pwrname = '\"' + Winname +'\"'
    cmd = ['powershell', 'compress-archive', Winfile ,  pwrname
]
    subprocess.run(cmd)
#Powershell compress-archive -Path String -DestinationPath
String
    Winname = os.path.join(subdir, Winname)
    CompMethod = 'Windows Compress '
    EndRealTime = time.time()                        #End Timer
    if os.path.isfile(Winname):                      #windows
compress archive fails when used through powershell on large
files. This is for error catching.
        ENDFileSize = os.path.getsize(Winname)
#get new file size (in bytes)
        W=1
        WinS = ENDFileSize
        Windata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
    else:
        Windata = str(CompMethod) + t + 'FAILURE'
#record errors
#end Windows compress-archive test
################################################################
######################################
#begin PPMd
    FilePath = os.path.join(subdir, file)
    RealTime = time.time()
#Take start time for Data collection
    PPMdnamestart=file+'PPMdtest.7z'
#setup file name for PPMd test
    PPMdname = os.path.join(subdir,PPMdnamestart)
    cmd = ['7z', 'a', PPMdname , FilePath, '-m1=PPMd']
    subprocess.run(cmd)
#Run Deflate command
    CompMethod = '7zip: PPMd '
    EndRealTime = time.time()                              #End Timer
    ENDFileSize = os.path.getsize(PPMdname)
#get new file size (in bytes)
    PPMdS = ENDFileSize
    PPMddata = str(CompMethod) + t + str(OGFileSize) +t
+str(ENDFileSize)+ t+ str(ENDFileSize/OGFileSize)+ t+
(pathlib.Path(file).suffix) +t +str(EndRealTime -
RealTime)#compile data collected to be written to a file
#End PPMd
################################################################
######################################
#Record which file had the best compression ratio and delete
other files:
    if(W==1):
        if(MXS <= DeflateS and MXS <=WinS and (MXS<=OGFileSize
or os.path.isdir(FilePath)) and MXS <=PPMdS):
            os.remove(Deflatename)
            os.remove(Winname)
            os.remove(PPMdname)
        elif(DeflateS <=MXS and DeflateS <=WinS and
(DeflateS<=OGFileSize or os.path.isdir(FilePath)) and DeflateS
<=PPMdS):
            os.remove(MXname)
            os.remove(Winname)
            os.remove(PPMdname)
        elif(WinS<=MXS and WinS <=DeflateS and (WinS<=OGFileSize
or os.path.isdir(FilePath)) and WinS <=PPMdS):
            os.remove(MXname)
            os.remove(Deflatename)
            os.remove(PPMdname)
        elif(PPMdS<=MXS and PPMdS<=DeflateS and PPMdS <= WinS
and (PPMdS <=OGFileSize or os.path.isdir(FilePath))):
            os.remove(MXname)
            os.remove(Deflatename)
            os.remove(Winname)
        else:
            os.remove(MXname)
            os.remove(Deflatename)
            os.remove(Winname)
            os.remove(PPMdname)
    elif(W==0):
        if(MXS<=DeflateS and MXS<=PPMdS and (MXS<=OGFileSize or 
os.path.isdir(FilePath))):
            os.remove(Deflatename)
            os.remove(PPMdname)
        elif(DeflateS<=MXS and DeflateS<=PPMdS and (DeflateS 
<=OGFileSize or os.path.isdir(FilePath))):
            os.remove(MXname)
            os.remove(PPMdname)
        elif(PPMdS<=MXS and PPMdS <= DeflateS and (PPMdS 
<=OGFileSize or os.path.isdir(FilePath))):
            os.remove(MXname)
            os.remove(Deflatename)
        else:
            os.remove(MXname)
            os.remove(Deflatename)
            os.remove(PPMdname)

#end cleanup
################################################################
######################################
#write results to data file
    f= open('C:\Capstonedata\data.txt', 'a')
    f.write('\n')
    f.write(MXdata)
    f.write('\n')
    f.write(Deflatedata)
    f.write('\n')
    f.write(Windata)
    f.write('\n')
    f.write(PPMddata)
    f.close()
#Write complete
    return() #end of compression function
```

```
dir_list = os.listdir()
# Get the list of all files and directories in the root
directory
rootdir = os.getcwd()
#setup data file
t= ';'
f= open('C:\Capstonedata\data.txt', 'a')
f.write('\n')
f.write('CompMethod')
f.write(t)
f.write('Original file size')
f.write(t)
f.write('compressed file size')
f.write(t)
f.write('compression ratio')
f.write(t)
f.write('File Type')
f.write(t)
f.write('real time')
f.close()
#create array to store[total data amount, total data compressed
as LZMA2, total data as Deflate, total data as win, total data
as PPM-whatever , and total data when taking the best result]
import array as arr
array = arr.array('l', [0, 0, 0, 0, 0, 0])
print('enter "T" for compression test, enter "C" to compress
files')
Input= input()
if(Input == 'T'):
# Test all files in all subdirectory individually
    for subdir, dirs, files in os.walk(rootdir):
#Walkthrough entire current working directory
        for file in files:
            print(os.path.join(subdir, file))
            array = compresstest(subdir,file,array)
    f= open('C:\Capstonedata\data.txt', 'a')
    string = '\n' + 'Total bytes tested: '+ str(array[0])
+'000'+ t + 'Total after LZMA2 compression: ' + str(array[1])
+'000'+ t +'Total after Deflate compression: ' +
str(array[2])+'000' + t + 'Total after windows compression: '+
str(array[3]) +'000' +t + 'Total after PPMd compresswion: ' +
str(array[4])+ t + 'Total bytes with best compression selected:
' + str(array[5])+'000'
    f.write(string)
#record data from 'array' to the data.txt file
    f.close()
elif(Input =='C'):
#Practical application of the program
    print('Enter "S" to compress all contained files
individually, enter "F" to compress only files and folders in
the current directory')
    Input= input()
#user decides if they will do all files in all subdirectorie, or
only files and folders in the current working directory
    if(Input =='S'):
        for subdir, dirs, files in os.walk(rootdir):
            for file in files:
                print(os.path.join(subdir, file))
                compress(subdir,file)
    elif(Input == 'F'):
        for files in os.listdir(rootdir):
            print(files)
            compress(rootdir,files)
    else:
        print('invalid input. Script ending')
    f.close()
else:
    print('invalid input. Script ending')
```

Findings

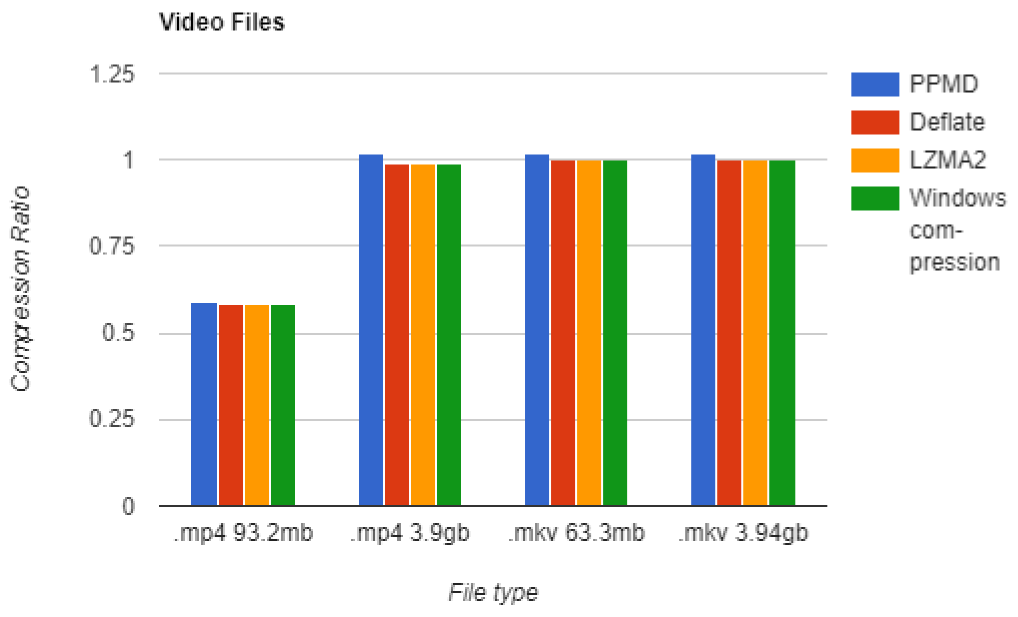
After performing the tests on the 16 different files, the program collected information about the original file size, compressed file size, compression ratio, file type, and real time. The results were not as drastically different as initially thought; the compression ratios were all fairly similar within each file. However, compression speeds did show more differences. The data has been made into 4 bar graphs, three graphs to show the Compression Ratios and two to show the Compression speed.

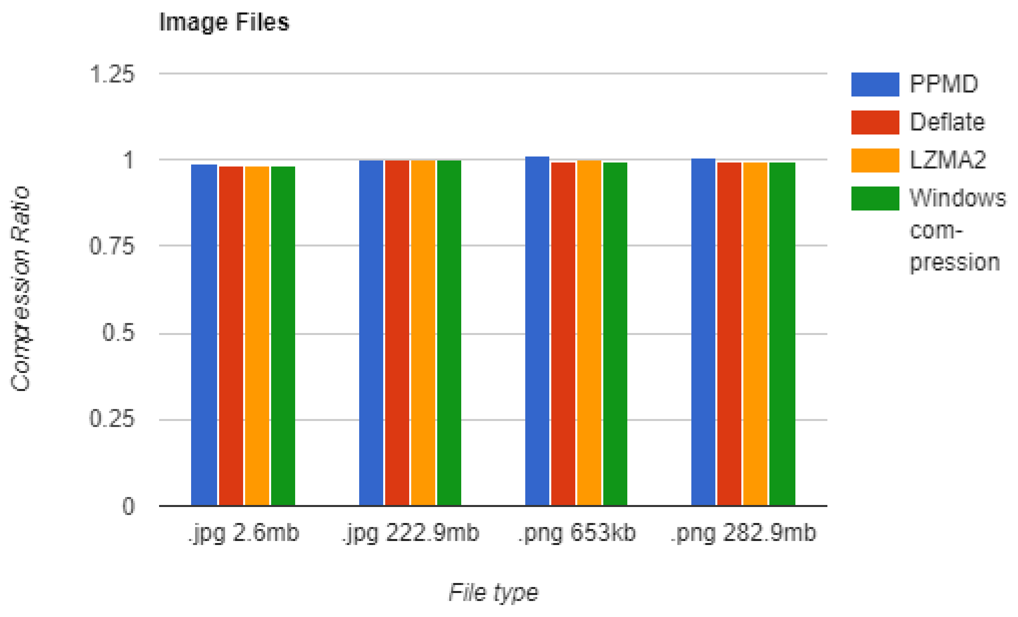
(Figure 1)



As the table shows (Figure 1), LZMA2 performed the best on all files except mp3. Windows Compress did the worst with compression ratio. However, the difference between the compression ratios is negligible; practically speaking, the slight difference in compression ratio is not going to make a difference. None of the compression methods used worked well to compress mp3 files.

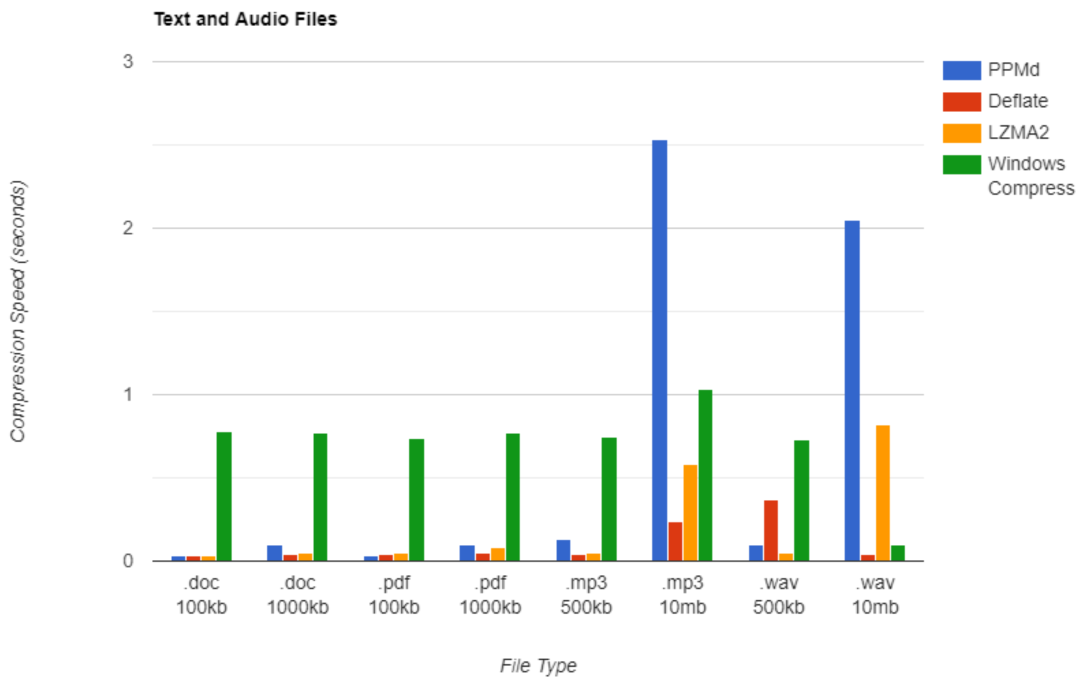
(Figure 2.1 & 2.2)





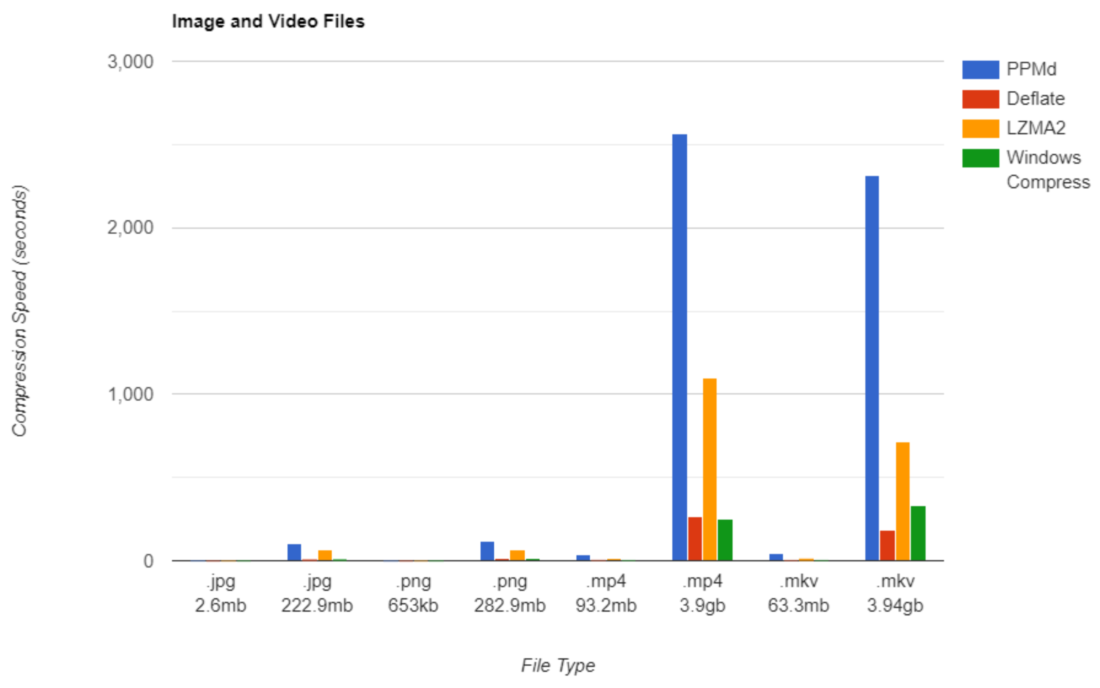
The image and video files did not compress well aside from the smaller MP4 file. The compression ratio for most of the video and image files was so close to 1 that it would not make sense to bother with compressing the files using any of the compression methods the program used. The data appears to show that smaller .MP4 files compress much better, but since there is only data for 1 small MP4 file no conclusions can be drawn about which method might be the better option. Rather, what the data shows is that these file types are already data rich, .mkv files in particular already contain compressed (tokenized) data.

(Figure 3)



In Figure 3 the difference in compression speed between the algorithms become clear. Windows compress tended to take longer on all of them except the audio files. Windows compress maintained a similar speed (excluding the 10mb .WAV file) for all the files despite changes in size and file type. Deflate maintained very high speeds for all of the files except the smaller .WAV file. Due to the fact that the compression ratios are very similar between all 4 compression methods, speed would probably be the determining factor in deciding what compression method to use for audio and text files.

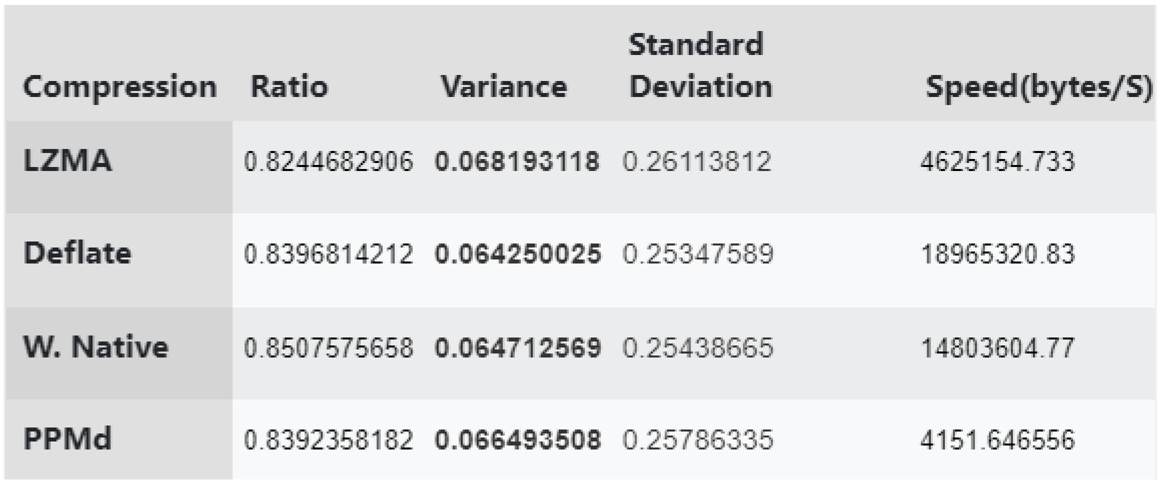
(Figure 4)



The graphic above (Figure 4) depicts the compression speed results for the image and video files. Large movie files took longer to compress which is understandable due to the size of the files. PPMd took the longest in the majority of the tests. Deflate and Windows Compress tended to take the least time and the results for them are very similar.

With all this being said, tests were only run on 16 files in total; this means some of the files could be outliers and were compressed better or worse then the compression method does on average. There are also many more file types that the tests could be run on; the results might drastically differ if the test was run on more then 8 file types. The sample size of files was too small to produce an accurate statistical analysis. More research would have to be done with the program to conclude anything about differences in speed or compression ratios.

(Figure 5)



Compression	Ratio	Variance	Standard Deviation	Speed(bytes/S)
LZMA	0.8244682906	**0.068193118**	0.26113812	4625154.733
Deflate	0.8396814212	**0.064250025**	0.25347589	18965320.83
W. Native	0.8507575658	**0.064712569**	0.25438665	14803604.77
PPMd	0.8392358182	**0.066493508**	0.25786335	4151.646556

Despite not being able to draw any concrete conclusions; the tests done showed that LZMA overall had the lowest compression ratio (Figure 5). However, this does not mean LZMA was the best algorithm by any means; all of the compression methods tested had an average compression ratio between 0.82 and 0.86. The difference between the compression ratios is so small that any storage space saved might be insignificant. Speed is where the algorithms have much more differences. Deflate and Windows native Compression performed significantly better than LZMA in terms of speed. Deflate was as much as four times faster at compressing data than LZMA, and a couple thousand times faster than deflate derivative algorithm PPMd.

It should however be noted that PPMd performed extremely well on text documents, which is its intended use case, and very poorly on video files often adding to the total file size rather than compressing them. Of the algorithms listed, only Windows native fails to excel in any category.

Data

CompMethod	Original file size	compressed file size	compression ratio	File Type	real time
7zip: LZMA2	1048576	389518	0.3714733124	.doc	0.0858566761
7zip: Deflate	1048576	402706	0.3840503693	.doc	0.0717158318
Windows Compress	1048576	405447	0.3866643906	.doc	0.7798640728
7zip: PPMd	1048576	399532	0.381023407	.doc	0.2052974701
7zip: LZMA2	102602	78103	0.7612229781	.pdf	0.0638287067
7zip: Deflate	102602	79816	0.7779185591	.pdf	0.0408904552
Windows Compress	102602	80348	0.7831036432	.pdf	0.785061121
7zip: PPMd	102602	78158	0.76175903	.pdf	0.0488703251
7zip: LZMA2	10580756	8131733	0.7685398851	.wav	1.96819067
7zip: Deflate	10580756	9062802	0.8565363382	.wav	0.5766518116
Windows Compress	10580756	9976769	0.9429164608	.wav	1.405020714
7zip: PPMd	10580756	8461055	0.7996645041	.wav	4.226637125
7zip: LZMA2	1052352	882844	0.8389246184	.pdf	0.131080389
7zip: Deflate	1052352	892694	0.8482846044	.pdf	0.0850019455
Windows Compress	1052352	896963	0.8523412318	.pdf	0.8584561348
7zip: PPMd	1052352	881459	0.8376085188	.pdf	0.5311334133
7zip: LZMA2	10494922	10492862	0.9998037146	.mp3	1.272985697
7zip: Deflate	10494922	10495354	1.000041163	.mp3	0.4911968708
Windows Compress	10494922	10488391	0.999377699	.mp3	1.334067822
7zip: PPMd	10494922	10580892	1.008191581	.mp3	5.366489649
7zip: LZMA2	512457	512233	0.9995628902	.mp3	0.0903208256
7zip: Deflate	512457	511825	0.9987667258	.mp3	0.046875
Windows	512457	511322	0.99778518	.mp3	0.8329119682

Compress					
7zip: PPMd	512457	519962	1.014645131	.mp3	0.2493326664
7zip: LZMA2	512764	389099	0.7588266727	.wav	0.0887629986
7zip: Deflate	512764	436058	0.8504068148	.wav	0.0540575981
Windows Compress	512764	475580	0.9274832086	.wav	0.8271255493
7zip: PPMd	512764	409828	0.7992526776	.wav	0.1695275307
7zip: LZMA2	102400	13624	0.133046875	.doc	0.0433294773
7zip: Deflate	102400	16336	0.15953125	.doc	0.0360116959
Windows Compress	102400	16732	0.1633984375	.doc	0.800481081
7zip: PPMd	102400	16980	0.1658203125	.doc	0.0359046459
7zip: LZMA2	2684320	2645336	0.9854771413	.jpg	0.2922184467
7zip: Deflate	2684320	2647658	0.9863421649	.jpg	0.1695458889
Windows Compress	2684320	2647660	0.9863429099	.jpg	0.8418259621
7zip: PPMd	2684320	2657644	0.9900622877	.jpg	1.168282747
7zip: LZMA2	233682756	233343046	0.998546277	.jpg	69.79938936
7zip: Deflate	233682756	233372664	0.9986730215	.jpg	9.287797451
Windows Compress	233682756	233322528	0.9984584742	.jpg	10.63634658
7zip: PPMd	233682756	233773613	1.000388805	.jpg	109.2361295
7zip: LZMA2	4231099911	4231053812	0.9999891047	.mkv	754.2296588
7zip: Deflate	4231099911	4229758596	0.9996829867	.mkv	227.3937159
Windows Compress	4231099911	4226535987	0.9989221736	.mkv	328.6571908
7zip: PPMd	4231099911	4312158914	1.019157903	.mkv	1982.003952
7zip: LZMA2	66375067	66347307	0.9995817707	.mkv	9.939540148
7zip: Deflate	66375067	66366479	0.9998706141	.mkv	2.543972254
Windows Compress	66375067	66351104	0.9996389759	.mkv	3.39
7zip: PPMd	66375067	67656918	1.019312237	.mkv	29.79238009

7zip: LZMA2	4202369733	4198185825	0.9990043932	.mp4	681.0875137
7zip: Deflate	4202369733	4200207052	0.9994853663	.mp4	217.8640432
Windows Compress	4202369733	4199043072	0.9992083845	.mp4	252.38
7zip: PPMd	4202369733	4285174024	1.01970419	.mp4	1862.409872
7zip: LZMA2	97733380	56930764	0.5825109497	.mp4	7.561374903
7zip: Deflate	97733380	57002950	0.583249551	.mp4	3.116908312
Windows Compress	97733380	57036800	0.5835959014	.mp4	3.13
7zip: PPMd	97733380	57835543	0.5917685749	.mp4	25.76409912
7zip: LZMA2	296664305	295549206	0.9962412094	.png	75.59111452
7zip: Deflate	296664305	295711637	0.996788734	.png	11.77773237
Windows Compress	296664305	295758599	0.9969470341	.png	13.09227514
7zip: PPMd	296664305	298534357	1.006303596	.png	130.1534107
7zip: LZMA2	668709	667867	0.9987408574	.png	0.1424427032
7zip: Deflate	668709	665549	0.9952744766	.png	0.0568482876
Windows Compress	668709	665992	0.9959369472	.png	0.7992825508
7zip: PPMd	668709	677476	1.013110337	.png	0.2852375507

Along with the data collected, other sources have performed similar tests to compare compression algorithms. A study (Kodituwakku and Amarasinghe. p.142-147) comparing lossless compression algorithms with text files found that run length encoding, although fast, had a high compression ratio and in some cases the initial file was smaller than the compressed file. According to the tests performed in the study the LZW algorithm took more time then run length encoding however the compression ratio was a lot lower. The study compared 5 algorithms and in their tests with text data, Run length encoding performed the worst and LZW performed the best on average.

Challenges

Creating a program to compare algorithms did have its challenges. One of which was working with WinRAR which was not able to be included in the program. The team did not find a way to compress files in python using WinRAR. To compare WinRAR compression ratios and times the program would likely have to work differently; instead of compressing the files within the program itself, the program would likely have to collect information while using the WinRAR application to compress the files.

If the program was going to continue to be developed, it would be expected that there would be more challenges. One such challenge would be expanding it to be able to use other compression algorithms, currently the program only works with 7z, and windows compression methods. Additionally in its current state the script has issues passing large files to powershell for windows native compression.

In order to overcome these challenges and flesh out the script to more completely analyze compression techniques the script would need to be able to handle WinRAR, record CPU processing time, and record data on file decompression.

WinRAR, and CPU processing time are actually somewhat the same problem. In order to make WinRAR's script work with python a module would need to be developed to integrate with the API. Integrating all of the existing compression algorithms into the script using modules would allow for the collection of data concerning how many instructions/ how long it takes for the CPU to work on tasks. So both of these challenges can be addressed the same way.

In terms of file decompression speed, it should be about as easy as the original script was. However, due to the fact that the earlier change would change how decompression would be accessed and called by the script the previous listed change would need to occur first.

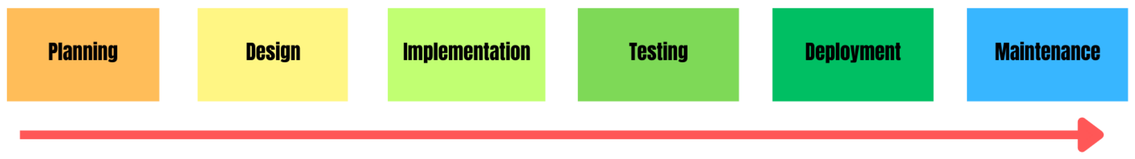
The biggest issue when collecting the data for a comparison was finding files large enough and small enough to compare. For each file type the files had to be a similar size and many did not get very big. It was fairly difficult to find large file sizes for text files. Since file types vary so drastically depending on what type of media they are for, files of similar size were selected for each media type.

Another challenge was the time/budget restraints; at the current state this program is very minimal. More time would be needed to expand and add more features. The project would also benefit from a bigger team working on the project. Currently the program costs nothing to run as it's only used on team members personal devices. Releasing an application to the public for people to use would come with expenses. Currently there is no budget for the application so no finished product could be pushed into production.

SDLC Plan

According to AWS there are 6 stages in the Software Development Life Cycle; The SDLC is a lifecycle that describes the process of how software is efficiently planned and created (Amazon, n.d).

(Figure 6)



The first stage is Planning. The software will have a maintenance plan, a troubleshooting plan, an approximate budget to maintain the software, a team to work on the project, and resources required. Design is the next stage, this is where the actual software development will happen. This is where developers will figure out how to create the front end and back end of the application. Developers will figure out what API's to use to time compression methods and collect information about compression ratios. Implementing is where the software gets programmed, this will be in the programming language Python and developers will use tools like IDE's and Github. Testing is when the team will check for problems in the software and fix any bugs found. The software will then enter the Deploy stage where a copy of it will be put into production. With Maintenance the team will manage the software and perform upgrades and fixes. There are different methodologies to creating software; for this project the team would most likely use a waterfall approach since the scope of the project is fairly small. The waterfall methodology is a good choice for smaller projects. The down side to the waterfall model is that it lacks flexibility since stages of the project have to be finished before starting the next one (Eye on Tech, 2022).

Implementation

The program would have to be first created in a development environment. Once the project reached the implementation stage, a copy of the program would be put into a production environment.

Implementing this software will require resources. Hosting the application in the cloud is how the team plans to save on costs. Instead of paying for physical hardware and spending time maintaining it the application will take advantage of cloud resources. Using a Cloud Service Provider to run applications also has many advantages to physical machines. First, Using a CSP would mean the team is only paying for what the application uses. Additionally, no one would have to be hired to maintain servers to run the application. Amazon Web Services offers a variety of services to aid in hosting the application. Amazon's Compute services offer virtual machines called EC2 instances. AWS also has database services for relational databases and non relational databases. There are multiple storage services in AWS that can provide different types of storage. Two types of storage that Amazon offers are block storage and object storage.

The application will need to have a team dedicated to monitor the application, maintain it, develop and release updates, and debug. Developers will get to share ideas and collaborate to improve the software. The team will consist of 3 people; First a back-end and front end developer will be needed to update the application. A debugger will also be hired to fix bugs in the code. The average salary for a back end developer is $103, 394. The average salary for a front end developer is $97, 313. Finally the average salary for a debugger is $115, 270.

The expected cost to maintain this program yearly is $350,977. This number was calculated based on the salary of the team which would be around $315,977. The approximate cost to build a SaaS application in the AWS cloud was also factored in. The application is fairly basic so the cost would be about $30,000-$35,000 according to one source (Srivastava, 2023).

Maintenance Plan

The software will be supported for 2-years after a final deliverable has been presented. Different types of updates that will need to be performed on this software. How updates are performed will depend on the importance of the update. Every update, excluding emergency fixes, will require a request and will be documented thoroughly.

The application will be hosted in AWS and Amazon CloudWatch will be set up to monitor the application and detect problems. Thresholds will also be set up with regards to traffic to the application, costs incurred, and security; if a threshold is triggered an alarm will be set to the email of the person responsible for handling that area of the application. This application will require an elastic load balancer to scale resources depending on site traffic. AWS accounts to manage this application will have passwords and a password policy will be set. Passwords will be at least 10 characters, containing an uppercase letter, a lowercase letter, a number, and a symbol. These passwords would also have to be updated every 6 months to keep the application secure. The root account will be used for access management, it will not be used for day to day operations. Redundancy will also be implemented to ensure data won't be lost and to keep high availability on the application.

If serious problems are detected through AWS monitoring services, the problem will be immediately handled. Problems relating to user experience or data security will be top priorities when the application is released into production. The team wants to make sure consumers using the site have the best experience possible and know that their information is safe. The list of problems that will be considered an emergency and will need to be fixed immediately are:

1. Security Threats
2. High Latency
3. Application Health
4. Long periods of down time
5. Network/Instance/Storage Failure

There will also be regular updates to the application and cloud environment to try and mitigate issues. At the start of every month an analysis will be conducted on features that could be added. New features will be integrated at the end of the month. Github will be used for version control and pushing these new features. Performance fixes will not wait till the end of the month and will be deployed immediately.

Troubleshooting Plan

Inevitably this software will have problems and fixes that will need to be made. There will also need to be a plan to help users work through problems using the application.
In the application there will be an email address for users to contact about issues with the site. Someone will review these issues and determine if there is something wrong with the site. If there is, the issue will be sent to the Debugger to identify the problem and fix it. If the issue is user error someone will contact the person about how to resolve the issue they are facing. The application will also have an FAQ where frequently asked questions with answers can be written to help guide users through using the software. In general the team will follow CompTIA's six step methodology for troubleshooting.

The six steps are (Garn, 2021) :

1. "Identify the Problem"
2. "Establish a Theory of Probable Cause"
3. "Test the Theory to Determine the Cause"
4. "Establish a Plan of Action and Implement a Solution"
5. "Verify Full System Functionality and Implement Preventive Measures"
6. "Document Findings"

While creating the software, errors will be identified by testing individual parts of the program to narrow down where the error might be. Once the error is found it will be corrected. The debugger will also be responsible for identifying and fixing bugs in the program. Documentation will also play an important role in troubleshooting. In the code comments will be left throughout when developing to determine what parts of the code do. Documentation will also be created and saved whenever problems occur. This documentation can help find solutions to future problems faster by looking back at old cases and how they were solved. Documentation will be stored securely and organized by the date of when the problem was first recorded.

There are also different diagnostic tools to help identify and fix problems. For issues with the cloud, AWS offers lots of diagnostic services for applications hosted in their services. There are also third party cloud diagnostic services that can be implemented in the cloud.

Moving Forward

If research was to continue to compare more lossless compression methods and more file types the programs used would have to be updated. Currently the program only works with 4 compression algorithms. Since there are many more compression methods out there, actually producing a deliverable that worked with a good amount of compression algorithms would take more time. The other Problem is that Python is limited in the number of compression methods it can use. It can compress LZMA, DEFLATE, windows native compression, and PPMd.

It would also be helpful to compare decompression speeds. To do this the program would need some way of decompressing the files. The timer the python program used would run while the decompression happened so data could be collected. This information would be helpful because most users want their compressed files to decompress at high speeds so they can be accessed quicker. Decompression speed is ultimately just as important as compression speed.

To make this program more user friendly, a graphical user interface could be created instead of a CLI; Python has a module named Tkinter which is designed for creating GUI's. Another thing that could be added to the program is a method to sort the data obtained. For example, instead of a long list of all the files and how fast they compressed and their compression ratio, data could be sorted by which compression type was used from smallest to largest file size; this would make it much easier to compare the data collected.

A web application could be created that's more focused on picking the best lossless compression method for a user. This program could ask a user what file type they need compressed, the file size, volume of files, and if speed or compression ratio is more important for them. The program could then take the data from the original program and use it to determine the best algorithm.

The mission with regard to security is to make sure no one can access data produced by the application or any data pertaining to the application. To secure this application, firewalls could be put in place and a security monitoring system could be set up. The Principle of least privilege would also be used to ensure anyone working on the project only had exactly what they needed to do their job. Encryption could also be implemented to protect information. Regular maintenance would have to be performed on this application to keep it up to date and secure. It would have to go through updates to add features and to debug the program.

Ultimately the program as it stands currently exists more as a research tool, and less as a practical way to increase storage. While there are some easy changes that can be made to make it practical for the purpose of storing and compressing data efficiently, more research would need to be done to make it as efficient as a human carefully managing their data manually. However the program could see niche applications working with deep storage/ offline storage where individual files need to be easily accessible.

In order to make the program the truly robust piece of software it would need to be in order to be considered beneficial it would need to be transferred to another language, and integrate with 7zip API.

Conclusion

Lossless compression is a great way for users to store data they do not access frequently; lossless compression can save lots of space while at the same time retaining all the data the original file had. Lossless compression is built into many common file types so that computers can store more files. However, as shown, these files can be even further compressed down, saving even more space.

This software solution can help people make the best of their storage space. The importance of this is undeniable. Expanding or upgrading storage can be expensive and difficult so it's important to use storage space as efficiently as possible. With this research tool it would make it easy to compare various compression methods. This comparison could help users choose a way to losslessly compress files. The compression algorithm a user chooses to use depends on multiple factors such as file type, file size, and other preferences. To maximize efficiency when compression files the algorithm chosen should have low compression and decompression speeds and a low compression ratio.

With more development to make the program more user friendly, the software solution presented could help determine which lossless compression algorithm is the best for users. This research tool could test and compare compression methods, making it easier to see the difference between lossless compression algorithms. While the initial program only compares Zip compression methods it can still be used as a research tool. This program needs a lot more work to make it user friendly. Currently the program is difficult to run and limited in the compression methods it offers.

In conclusion, implementing this software would take a team and require time and resources. A maintenance plan and a trouble shooting plan would have to be implemented to make sure the application runs smoothly. There would need to be a fairly large budget assuming the team would hire employees. If the team were to just run the application without recruiting other developers it would likely cost around $35,000 to run the application in a AWS cloud environment.

References

"Application and Infrastructure Monitoring – Amazon Cloudwatch – Amazon ..." *AWS*, Amazon Web Services, https://aws.amazon.com/cloudwatch/.

"Application Hosting." *Amazon*, Amazon, https://aws.amazon.com/application-hosting/.

Blelloch, Guy E. "Introduction to data compression." *Computer Science Department, Carnegie Mellon University* (2001): 54.

Buenavida, Leah Fainchtein. "Crunch Time: 10 Best Compression Algorithms - Dzone." *Dzone.com*, DZone, 28 May 2020, https://dzone.com/articles/crunch-time-10-best-compression-algorithms.

Chai, Wesley, and Ivy Wigmore. "AWS CloudWatch." *Techtarget.com*, TechTarget, 17 May 2021, https://www.techtarget.com/searchaws/definition/CloudWatch#:~:text=CloudWatch%20a nd%20CloudTrail%20are%20both,environment%20through%20tracking%20API%20call s.

"Data Compression." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2023, https://en.wikipedia.org/wiki/Data_compression.

Ethw. "History of Lossless Data Compression Algorithms." *ETHW*, ETHW, 22 Jan. 2019, https://ethw.org/History_of_Lossless_Data_Compression_Algorithms.

Feldspar, Antaeus. "An Explanation of the DEFLATE Algorithm." *An Explanation of the `Deflate' Algorithm*, 23 Aug. 1997, https://www.zlib.net/feldspar.html.

Garn, Damon M. "Use a Troubleshooting Methodology for More Efficient It Support." *CompTIA*, CompTIA, 6 Dec. 2021, https://www.comptia.org/blog/troubleshooting-methodology.

Gomez, Jose. "Software Maintenance Plan." *Koombea*, Koombea, 24 Apr. 2023, https://www.koombea.com/blog/software-maintenance-plan/.

Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes" (PDF). *Proceedings of the IRE*. **40** (9): 1098–1101.

Kapil, Archish Rai. "What Are Data Compression Techniques?: AnalytixLabs." *Blogs & Updates on Data Science, Business Analytics, AI Machine Learning*, 25 July 2022, https://www.analytixlabs.co.in/blog/data-compression-technique/.

Ketshabetswe, Keleadile Lucia, et al. "Data compression algorithms for wireless sensor networks: A review and comparison." *IEEE Access* 9 (2021): 136872-136891.

Kodituwakku, S. R., and U. S. Amarasinghe. "Comparison of lossless data compression algorithms for text data." *Indian journal of computer science and engineering* 1.4 (2010): 416-425.

“Lossless Compression: A Complete Guide | Adobe.” *Adobe.com*, Adobe, https://www.adobe.com/uk/creativecloud/photography/discover/lossless-compression.htm l.

Matthew. “Building Secure Applications: Top 10 Application Security Best Practices.” *Sqreen Blog*, Sqreen, 21 Apr. 2021, https://blog.sqreen.com/best-practices-build-secure-applications/.

“Mission Transcripts - Apollo 17.” *NASA*, NASA, https://historycollection.jsc.nasa.gov/JSCHistoryPortal/history/mission_trans/apollo17.ht m.

Muhammadumar. “Compression Algorithms – a Brief Compendium.” *File Format Blog*, 3 Sept. 2021, https://blog.fileformat.com/2021/09/03/lossy-and-lossless-compression-algorithms/.

Porwal, Shrusti, et al. "Data compression methodologies for lossless data and comparison between algorithms." *International Journal of Engineering Science and Innovative Technology (IJESIT) Volume* 2 (2013): 142-147.

“Prediction by Partial Matching.” *Wikipedia*, Wikimedia Foundation, 31 Jan. 2023, https://en.wikipedia.org/wiki/Prediction_by_partial_matching#:~:text=Prediction%20by %20partial%20matching%20(PPM,next%20symbol%20in%20the%20stream.

Reghbati, Hassan K. "Special feature an overview of data compression techniques." *Computer* 14.04 (1981): 71-75.

“Salary: Back End Developer (March, 2023) United States - ZipRecruiter.” *ZipRecruiter* , https://www.ziprecruiter.com/Salaries/BACK-END-Developer-Salary.

“Salary: Frontend Developer (March, 2023) United States.” *ZipRecruiter*, https://www.ziprecruiter.com/Salaries/Frontend-Developer-Salary.

“Salary: Software Debugger (March, 2023) United States - ZipRecruiter.” *ZipRecruiter.com*, https://www.ziprecruiter.com/Salaries/Software-Debugger-Salary.

Sample Pages of the Template for a Software Maintenance Plan - Techstreet. https://images.techstreet.com/direct/SWM_samples.pdf.

Srivastava, Sudeep. "How Much Does It Cost to Build a SAAS Application on AWS?" *Appinventiv.com*, Appinventiv, 31 Mar. 2023, https://appinventiv.com/blog/cost-to-build-saas-application-on-aws-cloud/.

"Storage." *Amazon Whitepaper*, Amazon, 2023, https://docs.aws.amazon.com/whitepapers/latest/aws-overview/storage-services.html.

Synoptek. "What Is Application Development and Maintenance?" *Synoptek*, Synoptek, 19 Dec. 2022, https://synoptek.com/insights/it-blogs/application-development-maintenance/#:~:text=What%20is%20Application%20Maintenance%3F,the%20best%20of%20their%20abilities.

"Thinking in Terms of Failure Modes." *Amazon*, Amazon, https://docs.aws.amazon.com/whitepapers/latest/aws-outposts-high-availability-design/thinking-in-terms-of-failure-modes.html.

Wang, Jie, and Zachary A. Kissel. "Data Compression Using ZIP." *Introduction to Network Security: Theory and Practice*, Wiley, Hoboken, NJ, 2015, pp. 381–382.

"What Is Run-Length Encoding (RLE)?" *Api.video*, Api.video, https://api.video/what-is/run-length-encoding.

"What Is SDLC (Software Development Lifecycle)?" *Amazon.com*, Amazon, https://aws.amazon.com/what-is/sdlc/.

"[MS-WUSP]: LZ77 Compression Algorithm." *Microsoft*, Microsoft, 10 Apr. 2023, https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb.