CPE 593 Final Project – File Compression

Joshua Canlas
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ
jcanlas1@stevens.edu

David Krauthamer
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ
dkrautha@stevens.edu

John Theising
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ
jtheisin@stevens.edu

***Abstract*—This project aims to implement and compare three different file compression algorithms: arithmetic coding, Huffman coding, and bzip2. The final results will compare how each algorithm performs using the following metrics: time to compress and decompress text files, and compressed file size. Each algorithm implemented in this project is used in modern-day file compression and gave the team an opportunity to learn more about the use cases of the different algorithms.**

I. Introduction

A. Arithmetic Coding

Arithmetic coding is a lossless algorithm and works by taking a symbol and assigning it a frequency to a table. However, instead of using the frequency table, arithmetic coding turns the frequency table into a probability model, which is used for encoding and decoding messages. There are two stages to the algorithm: calculating sub-intervals using floating point ranges, and using a binary search to find a binary sequence that lies within the final floating point sub-interval. In the first stage, each symbol in the message takes a sub-interval in the number line (0.0 to 1.0), corresponding to its probability. As the encoding process continues, the sub-intervals become smaller and smaller, resulting in high-precision decimals. Once each symbol goes through the encoding process, the entire message is encoded using a single decimal number between 0.0 and 1.0.

For the second stage, a binary sequence that is encapsulated within the last sub-interval from the previous stage is found using a binary search-like algorithm. The binary search begins with the 0.0 to 1.0 number line and is halved until a binary sequence is found that is within the last sub-interval from the previous stage. For a more detailed explanation and examples of the algorithm, see [1] [2]. An example will be shown later in this document (see Section II-A2).

An issue that arithmetic coding inevitably encounters is dealing with the limited precision that data types in programming languages can represent. Nelson introduces finite-precision arithmetic encoding by using unbounded precision math with integers [3]. This technique is used to ensure that files with a large number of symbols can be encoded without being limited to the precision of value types.

Another solution to solve the limited precision problem is by using modules (specifically in Python) that allow for higher precision. Gad introduces the *decimal* Python module, which allows for user-defined precision [2]. According to the module documentation, for a 64-bit system, the maximum precision can be 999999999999999999 [4], which should be sufficient for the implementation of this project.

B. Huffman Coding

Huffman coding is a lossless data compression technique that assigns variable-length codes to symbols based on their frequency. This means that frequently occurring symbols are represented by shorter codes, while less frequent symbols are represented by longer codes. Codes are represented as a sequence of bits, and are chosen such that they are ”prefix-free,” meaning the process for decoding symbols is unambiguous. Huffman coding follows the following steps:

1) Frequency analysis: The algorithm first analyzes the data to determine the frequency of each symbol.
2) Building the Huffman tree: The algorithm then builds a binary tree known as the Huffman tree. The leaves of the tree represent the individual symbols, and the internal nodes represent bits a particular code will consist of. The nodes are arranged so that the most frequent symbols are closer to the root of the tree.
3) Assigning codes: Each node in the Huffman tree is assigned a code. The code for a leaf node is the path from the root of the tree to the leaf, where a 0 represents a left branch and a 1 represents a right branch.
4) Encoding: The data is then encoded using the Huffman codes. Each symbol is replaced by its corresponding code, resulting in a compressed representation of the data.
5) Decoding: The compressed data can be decompressed by using the Huffman tree. The decoder starts at the root of the tree and follows the path determined by the next bit in the code. When a leaf node is reached, the corresponding symbol is decoded.

Huffman coding is optimal when encoding symbols separately [5], but better compression ratios can be obtained when symbols are grouped together with other algorithms.

C. bzip2

Bzip2 is a lossless compression algorithm that involves data transformations before Huffman encoding. The purpose of the data transformations is to turn the data into a more compressible format. The steps involved in bzip2 are:

1) Run Length Encoding
2) Burrows Wheeler Transform

3) Move-to-front transformation
4) Run Length encoding again
5) Huffman encoding [6]

The end result is a binary code that represents the original input string.

1) Run Length Encoding: RLE is used on the original input string to reduce the size of long runs of characters. The first 3 characters of the repeated letters are maintained, while any repeated letters after that are represented with a byte as to how many there are [6]. The motivation for the implementation of this is to reduce the size of the string without losing information. Repeated letters are very common in text, images, and numeric data, so RLE is very practical for this use. In practicality, this step is not necessary for the bzip2 encoding as next steps take care of that problem. It was only added to take care of a string of completely repeated characters [7]. This implementation assumes that is not the case and forgoes the initial RLE.

2) Burrows Wheeler Encoding: BWT is a transformation of the data involving the rotations of the string. In the bzip2 implementation, the data is split into n blocks of size 100 to 900 kb, sorted by the blocks, and then BWT encoding is performed on each block [6]. In this implementation, a memory and time efficient way to implement 100 to 900 kb could not be achieved, so the block size is set at 6. The motivation for BWT is to group similar characters together for the following steps of the compression [6]. The decoding for BWT involves sorting the BWT, getting running totals of how many times each character has been seen, and then using those as indexes on the sorted string to construct the original [8]

3) Move to Front: MTF is a transformation on the data where the block is reordered and indexed to create a representation that is amenable to the next step of RLE [6]. In the MTF, the lowest character is taken and appended to the output. Next, the index of that character is popped from the alphabet and placed at the beginning. This is done for the entire length of the data [6]. The output is often long strings of 0's because of repeated letters which is very useful for the next step.

4) Run Length Encoding 2: Slightly different than the first instance of RLE. In this process, long strings of 0's are encoded as a value and the length of the 0's is encoded directly after that [6]. The run length encoding in this case only applies to the 0's. This makes sense because 0 is the only character that should have repeated values after the MTF transformation. In the original bzip2 encoding, the numbers are encoded with a specific formula using the binary representation of the count in combination with the assumption that the only character is 0 to allow for very concise encoding while never actually encoding 0 [6]. In this implementation, the count is reset to 0 if it exceeds 255, only allowing for runs of 255 characters in a row and 0's are encoded.

5) Huffman Encoding: Huffman encoding has been described in Section I-B.

II. Design and Algorithms

A. Arithmetic Coding

1) Pseudocode: The algorithms shown below outline the encoding and decoding process of arithmetic coding.

Algorithm 1 Arithmetic Encoding

```
function ENCODE
    Split the message into sub-strings
    for each sub-string do
        Generate the probability table for the sub-string
        Min ← 0
        Max ← 1
        for each symbol in sub-string do
            Range ← Max - Min
            Max ← Min + Range * Symbol_prob[high]
            Min ← Min + Range * Symbol_prob][low]
        end for
        Find the binary sequence contained within Min and Max
    end for
    Save the list of binary sequences, sub-string lengths, and frequency tables
```

Algorithm 2 Arithmetic Decoding

```
function DECODE
    for each encoded message do
        Get the corresponding frequency table and calculate the probability table
        Convert the binary encoded message into a decimal
        Min ← 0
        Max ← 1
        for range of the message length do
            for symbol, value in current probability table do
                Range ← Max - Min
                Sub-max ← Min + Range * value[high]
                Sub-min ← Min + Range * value[low]
                if Sub-min <= encoded message < Sub-max then
                    Append to the decoded message
                    Min ← Sub-min
                    Max ← Sub-max
                    break
                end if
            end for
        end for
    end for
```

2) Arithmetic Coding Example: The following example was taken from [1].

If the message that needed to be encoded was the word, "Hello", a probability table would have to first be created. (see Table I).

Character	Frequency	Probability
H	1	0.2
e	1	0.2
l	2	0.4
o	1	0.2

TABLE I. Frequency and probability table

The number line (as shown in Fig. 1) would then be divided into four intervals, where the range of each interval is the symbol's corresponding probability.

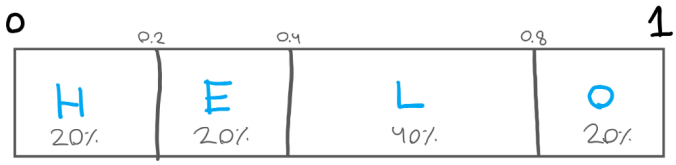

Fig. 1. Initial number line

The encoding starts with just the letter “H”, which would give the range of 0 to 0.2. Within this new number line, new intervals are created to fit inside it, using the same probability table in Table I. This is represented in Fig. 2

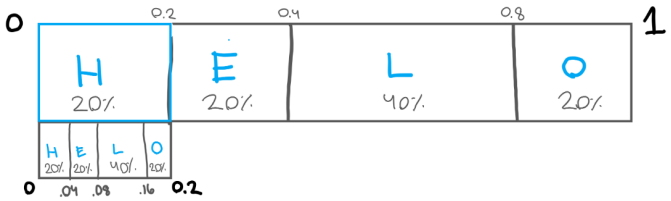

Fig. 2. Encoding the symbol ”H”

As the encoding process is repeated, this copying of the number line and fitting it within the previous range continues until the entire string is encoded. The final interval that encompasses the last character is used in the binary search part of the encoding process.

The binary search algorithm is done over a number line to find a binary range that lays within the range from the first stage (the decimal range). The number line (ranging from 0 to 1, initially) is laid out and is divided in half over and over again until the range falls within the decimal range that was calculated from the first stage. This is shown in Figs. 3 and 4.

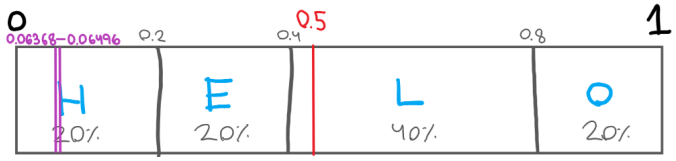

Fig. 3. Splitting number line in half to find binary sequence

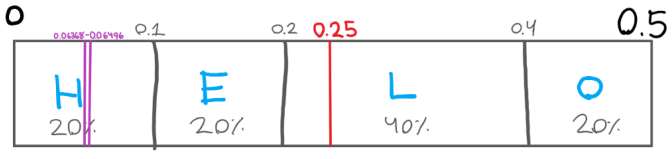

Fig. 4. Splitting number line in half again to find binary sequence

This splitting of the number line continues until we are left with a binary sequence that represents a target, just like the 0.5 and 0.25 from earlier examples, that lays within the encoded range from first stage. This binary stream is the coded version (the compressed version) of the entire message.

The decoder follows the same procedure as the encoder and should be able to decode the binary sequence, given the length of the entire message and the frequency table used to encode the message.

3) API Specifications: The algorithm is separated into two classes: ArithmeticEncoder and ArithmeticDecoder. It is important to note that this algorithm uses the *decimal* Python module to allow for more precision when calculating the sub-intervals. A number of helper functions used to load the text file and the encoded file have been listed as well. Note that the complexity of each function (big O) is listed next to each function.

ArithmeticEncoder

- member variables: encoded_msg_dict, precision, batch_size, encoded_msg, msg_len, prob_table_list, num_bits
- calc_freq_table(self, msg: str) -> Dict ($O(n)$)
- calc_prob_table(self, frequency_table: Dict) -> Dict ($O(n)$)
- encode(self, msg: str) -> None ($O(nmlog(n))$)

The member variables, precision, batch_size, and num_bits, were used to tune the algorithm to find the optimal compression ratio and speed. The precision refers to the precision of the decimal values used to calculate the sub-intervals for each symbol. Batch_size is the number of symbols that would be encoded for each encoding pass. Lastly, the num_bits is the number of bits that would be used for the binary sequence that represents the encoded message.

ArithmeticDecoder

- member variables: decoded_msg, precision
- bin2float(self, bin: List) -> Decimal ($O(n)$)
- decode(self, encoded_msg_list: List, msg_len: str, prob_table_list: List) -> str ($O(nml)$)

Helper Functions

- load_file(file_path: str) -> str ($O(n)$)
- load_encoded_msg(msg_path: str, helpers_path: str, precision: int) -> tuple[bitarray, str, Dict] ($O(n)$)

4) Modifications: Listed below are a couple of modifications that were implemented to get better compression time and ratio without using finite-precision encoding:

- Encode the entire message in smaller chunks
- Limit the binary sequence for each encoded message to 16 bits

B. Improved Huffman Coding

1) Base Implementation: The base implementation of Huffman Coding was done with Python. It utilized a third party library to handle arrays of bits [9] as well as a built in Python module to perform heap operations with a list [10].

The following psuedocode shows the algorithms utilized in the implementation:

Algorithm 3 Build Tree

function BUILDTREE
    $frequencies \leftarrow$ frequency of symbols in the source
    $pqueue \leftarrow []$
    for $data, count$ in $frequencies$ **do**
        $node \leftarrow$ NODE($count, data$)
        HEAPPUSH($pqueue, node$)
    end for
    while LEN($pqueue$) > 1 **do**
        $left \leftarrow$ HEAPPOP(pqueue)
        $right \leftarrow$ HEAPPOP(pqueue)
        $count \leftarrow left.count + right.count$
        $node \leftarrow$ NODE($count$, "", $left$, $right$)
        HEAPPUSH($pqueue, node$)
    end while
    return $pqueue[0]$
end function=0

Algorithm 4 Build Code

function BUILDCODE
    $code \leftarrow$ mapping from symbols to bits
    $node \leftarrow$ root node of the tree
    $arr \leftarrow$ string representation of bits
    if $node.data$ **then**
        $code[node.data] \leftarrow$ BITARRAY(arr)
        return
    end if
    if $node.left \neq null$ and $node.right \neq null$ **then**
        BUILDCODE($node.left$, $arr +$ "0")
        BUILDCODE($node.right$, $arr +$ "1")
    end if
end function

2) Word Grouping: The first improvement explored was to switch from using individual characters as symbols to entire words. The implementation of the basic Huffman encoder required no changes to be able to test this idea. The reason for this is that it operates on a list of symbols, and there is very little difference in iterating over a list of individual characters compared to a list of strings in Python. The reason for trying groups of words was the hope that the larger size of the Huffman tree would be offset by the space savings of condensing an entire word into just a few bits. Many words are longer than two characters (16 bits), so if the Huffman code for a word was smaller than this it could lead to a 50%+ reduction in size of the original source.

3) Hybrid Word Grouping: The second improvement explored was to combine using words and letters as symbols in a Huffman tree, with the hope that it would be a middle ground between individual letters and words. The algorithm used to build the tree is very similar to the one used for a typical Huffman tree, with the change that some number of the most frequently found words in the given input would also be put into the tree. The algorithm for building the codes for each node in the tree is identical to that of the previous implementation. The following are the three new functions implemented specifically for this hybrid approach:

- Build Frequencies: Takes the list of words and punctuation symbols and returns the frequency of each individual character and the top N most commonly present words.

- Build Tree: Uses the output of the previous function to add the words and characters to a priority queue, then builds a Huffman tree with the priority queue using the same algorithm as in the original implementation.

- Serialize: When encoding the source list of words, each word must first be checked for membership in the Huffman tree. If present, it will be encoded with its code, otherwise it will be encoded letter by letter.

Algorithm 5 Build Frequencies

function BUILDFREQUENCIES
    $source \leftarrow$ list of words and punctuation symbols
    $n \leftarrow$ number of most common words to include
    $onlywords \leftarrow []$
    for $symbol$ in $source$ **do**
        if LEN(symbol) > 1 **then**
            APPEND($onlywords$,$symbol$)
        end if
    end for
    $wordfrequencies \leftarrow$ freq of word in onlywords
    $cwords \leftarrow$ MOSTCOMMON($wordfrequencies$,n)
    $chars \leftarrow$ FLATTEN($source$)
    $charfrequencies \leftarrow$ freq of chars in chars
    return $cwords$,$charfrequencies$
end function

Algorithm 6 Build Tree Hybrid

function BUILDTREEHYBRID
    $cwords$,$charfrequencies \leftarrow$ BUILDFREQUENCIES()
    $pqueue \leftarrow []$
    for $data, count$ in $cwords$ **do**
        $node \leftarrow$ NODE($count, data$)
        HEAPPUSH($pqueue, node$)
    end for
    for $data, count$ in $charfrequencies$ **do**
        $node \leftarrow$ NODE($count, data$)
        HEAPPUSH($pqueue, node$)
    end for
    while LEN($pqueue$) > 1 **do**
        $left \leftarrow$ HEAPPOP(pqueue)
        $right \leftarrow$ HEAPPOP(pqueue)
        $count \leftarrow left.count + right.count$
        $node \leftarrow$ NODE($count$, "", $left$, $right$)
        HEAPPUSH($pqueue, node$)
    end while
    return $pqueue[0]$
end function

Algorithm 7 HybridSerialize

```
function SERIALIZE
    bits ← BITARRAY()
    for word in sourcewords do
        if word in code then
            codebits ← GET(code, word)
            EXTEND(bits, codebits)
        end if
    end for
    return bits
end function
```

C. bzip2

1) API Specifications: This implementation of Bzip2 is meant to be used strictly for compressing and decompressing files. It has been tested with .txt, .jpg and .tiff. The .jpg and .tiff are already compressed, so it results in expansion. However, the .NEF (.RAW) file was compressed.

There are two main functions run_bzip2_compression() and run_bzip2_decompression().

Main Functions

- *run_bzip2_compression(file_path, filename='test_file', block_size_kb=6, returned=False)* ($O(n^2)$)
    - *file_path*: the path the the file to be compressed
    - *file_name*: the prefix for the output files
    - *block_size_kb*: the block size for the BTW in kb. (Do not exceed 20 unless runtime and memory issues are not a problem).
- *run_bzip2_decompression(file_path, returned=False)* ($O(n * m)$)
  m is at most 255. This will return the original file.
    - *file_path*: the original filename
- *run_clean_up(file)*
  Deletes the files created by run_bzip2_compression. Can either specify a file or if none specified, will remove for all sample files.

Internal Functions

- *read_file(file_path)*
  Reads in a file specified in file_path with utf-8 encoding.
- *filter_to_ascii(data)*
  Filters a string to ascii characters.
- *split_into_blocks(data, block_size_kb)*
  Splits data into equal sized blocks.
    - block_size_bytes = *block_size_kb* * 1024
- *run_length_encode(data)*
  Not implemented in the compression, but here for completeness.
    - Example input = 'aaaaaabbccddddddd'
      Example output = aaaa$2#bbccdddd$3#
- *run_length_decode(data)*
  Iterates through a string and keeps count of consecutive repeated characters to represent as a number of repeats.
    - Example:
      Input = aaaa$1#bbccdddd$3#
      Iteration 1-4
      Output = aaaa
      Iteration 4-6
      Count_string = 1
      Iteration 7
      Output_to_append = 2*a
      Output = aaaaaa
- *generate_rotation_matrix(data)*
  Generates all rotations of string data. (see Fig. 5).

p	!	a	a	a	p	a
a	p	!	!	y	a	p
p	a	p	y	a	y	a
a	p	a	p	!	a	y
y	a	p	a	p	!	a
a	y	a	p	a	p	!
!	a	y	a	p	a	p

Fig. 5. *generate_rotation_matrix* function visual

- *burrows_wheeler_encode(data)* ($O(n^2)$)
  Generates the rotation matrix. The function also returns the index of the beginning character of the string in the sorted rotation matrix for decoding. (see Fig. 6).

!	a	y	a	p	a	p
a	p	!	!	y	a	p
a	p	a	p	!	a	y
a	y	a	p	a	p	!
p	!	a	a	a	p	a
p	a	p	y	a	y	a
y	a	p	a	p	!	a

Fig. 6. burrows_wheeler_encode function visual

- *burrows_wheeler_decode(data,original_index)* ($O(nlog(n))$)
  The proof for this approach is listed in the original paper [8]. In the paper, it only takes one pass through the list after a quicksort, but it is implemented with 2 passes in this approach.
- *mtf_encode(data)* ($O(n)$)
  The move to front algorithm is to cause long strings of 0's in the data.
    - A simplified example with index 1 to 4:
      Input = 'bbbaaccccd', alphabet = ['a','b','c','d']
      Iteration 1

Index = 2, letter = b, output = 2, alphabet = ['b','a','c','d'], output = 2
Iteration 2-3
Index = 0, letter = b, output = 0, alphabet = ['b','a','c','d'], output = 2,0,0
Iteration 4
Index = 2, letter = a, alphabet = ['a','b','c','d], output = [2,0,0,2]
... final_output = [2,0,0,2,0,3,0,0,0,4]

- *mtf_decode(data)* ($O(n)$)

- *run_length_encode2(data)* ($O(n)$)
  This is a similar version of the run_length_encode, but only for 0's.

- *run_length_decode2(data)* ($O(n)$)
  Similar to the run_length_decode again, but with some changes.

Huffman Coding $O(nlog(n))$
This is previously described, so there will be less detail for this section.

- *build_huffman_tree(frequency_dict)*

- *build_encoding_dict(node,code='', encoding_dct=None)*

- *huffman_encode(text)*

- *huffman_decode(text, huffman_dct)* ($O(n * m)$)
  m is at most 255

2) Pseudocode: The following pseudocode outline the algorithms that make up the bzip2 algorithm.

Algorithm 8 bzip2 Compression
function RUN_BZIP2_COMPRESSION
Initialize empty lists for all items to be saved
Read in the data
Parse and save the filename
Turn data into bytearray
blocks ← split_into_blocks(data,block_size_kb)
#run_length_encode(blocks) – commented out
for block in blocks **do**
burrows_wheeler_encode
Append the index to list of bwt indices
mtf_encode
run_length_encode2 (for the 0's)
huffman_encode
Append the huffman_dictionary to list of huffman_dictionaries
Append huffman_encoded result to output
Append the length of the huffman encoded result to list
Optional - keep a running total of bits with calculate_bits(huffman_encoded)
end for
Save the huffman_encoded_result
Save the list of huffman dictionaries
Save the list of huffman encoded result lengths
Save the burrows wheeler indices
if returned **then**
Return original data for comparison
end if

Algorithm 9 bzip2 Decompression
function RUN_BZIP2_DECOMPRESSION
Open huffman_dictionary file
Length of encoded result file for indexing
Open the encoded result
Open the burrows wheeler indices file
Open the file header
for i, bw_index in enumerate(burrows_wheeler_indices) **do**
block ← blocks[:length_of_block_list[i]]
huffman_decode(block, huffman_dictionary_list[i])
run_length_decode(huffman_decoded)
mtf_decode(run_length_decoded)
burrows_wheeler_decode(mtf_decoded, bw_indices_list[i])
Append to output
end for
#run_length_decode(output) – commented out
Write to output with '_decoded' as a suffix.

Algorithm 10 Run Length Encode (RLE)

```
function RUN_LENGTH_ENCODE
    count ← 1
    output_str ← ”
    for character in each character do
        if the next character is repeated then
            count is incremented by 1
        else if count is greater than 5 then
            Append escape_character_1 + count + escape_character2
            Reset the count
        else
            append count ∗ character to the output output_str is incremented by data[i]
        end if
    end for
    Handle the edge case for repeated characters at the end
    if count is less than 4 then
        Append 4 ∗ character + escape_char1 + str(count − 4) + escape_char2
    else
        encoded_data ← count ∗ character
    end if
```

Algorithm 11 Run Length Decode

```
function RUN_LENGTH_DECODE
    count ← 1
    output_str ← ”
    for character in each character do
        if the next character is repeated then
            count is incremented by 1
        else if count is greater than 5 then
            Append escape_character_1 + count + escape_character2
            Reset the count
        else
            append count ∗ character to the output output_str is incremented by data[i]
        end if
    end for
    Handle the edge case for repeated characters at the end
    if count is less than 4 then
        Append 4 ∗ character + escape_char1 + str(count − 4) + escape_char2
    else
        encoded_data ← count ∗ character
    end if
```

Algorithm 12 Generate Rotation Matrix

```
function GENERATE_ROTATION_MATRIX
    for i in data do
        beginning_of_string ← text[i:]
        end_of_string ← text[:i]
        Append (beginning_data + end_of_string) to output
    end for
```

Algorithm 13 Burrows Wheeler Encode

```
function BURROWS_WHEELER_ENCODE
    rotation_matrix ← generate_rotation_matrix
    sorted_rotation_matrix ← sort(rotation_matrix)
    for rotation in sorted_rotation_matrix do
        Append (rotation[-1]) to output
    end for
    index ← index_of_first_alphabetic_char_in_sorted_matrix
```

Algorithm 14 Burrows Wheeler Decode

```
function BURROWS_WHEELER_DECODE
    index_representation_of_bwt ← bytearray(len(bwt))
    sorted_bwt ← sorted(index_representation_of_bwt)
    Instantiate sorted_ocurrances (list) []*256
    Instantiate bwt_ocurrances (list) []*256
    Instantiate sorted_lst [] to represent the characters and how many times each has been seen as a tuple
    Instantiate bwt_index  for mapping the bwt indices to the sorted_bwt indices, bwt_index
    for i in range(len(bwt) do
        Increment sorted_ocurrances[sorted_bwt[i]] by 1
        Increment bwt_ocurrances[bwt[i]] by 1
        Append a tuple(sorted_bwt[i], cumulative occurrences) to the sorted(bwt_lst)
        bwt_index[(bwt_i, bwt_occurrances[bwt[i]])] ← i
    end for
    for _ in range(len(bwt) do
        correct_positional_char_and_count ← sorted_bwt[original_index]
        output[_] ← correct_positional_char_and_count[0] (only the character)
        original_index ← bwt_index[correct_positional_char_and_cour
    end for
```

Algorithm 15 Move to Front Encode

```
function MTF_ENCODE
    alphabet ← all_possible_bytes
    for symbol in data do
        index ← index of symbol in alphabet
        Append index to output
        Pop the index from the alphabet
        Prepend to front of alphabet
    end for
    Return output
```

Algorithm 16 Move to Front Decode

```
function MTF_DECODE
    alphabet ← all_possible_bytes
    for index in data do
        symbol ← alphabet[index]
        Append symbol to output
        remove symbol from alphabet
        Prepend alphabet with symbol
    end for
    Return output
```

Algorithm 17 Run Length Encode 2

```
function RUN_LENGTH_ENCODE2
    for symbol in data do
        if symbol is 0 then
            Increment count by 1
        else
            while count > 255 do
                Add 0 plus the count to the output
                Subtract 255 from the count
            end while
            if count > 0 then
                Add 0 and the count to output
                Reset the count to 0
            end if
            Add the symbol to the output
        end if
    end for
    while count > 255 do
        Add 0 and the count to output
        Subtract 255 from the count
    end while
    if count > 0 then
        Add 0 and the count to the data
    end if
```

Algorithm 18 Run Length Decode 2

```
function RUN_LENGTH_DECODE2
    i ← 0
    while i < len(data do
        symbol ← encoded_data[i]
        if symbol is 0 then
            count is the next character
            Add the char_at_first_index ∗ count to the output
            Increment i by 2 bytes
        else
            Add symbol to the decoded data
            Increment i by 1 byte
        end if
    end while
    Return decoded data
```

Algorithm 19 Huffman Encoding

```
function HUFFMAN_ENCODE
    for symbol in set(text) do
        frequency_dict[symbol] ← text.count(symbol)
    end for
    huffman_tree ← build_huffman_tree(frequency_dictionary)
    encoding_dict ← build_encoding_dict(huffman_tree)
    encoded_text ← bitarray()
    encoded_text.encode(encoding_dict, text)
```

Algorithm 20 Huffman Decoding

```
function HUFFMAN_DECODE
    for bit in data do
        Append bit to empty bitarray
        for symbol, code in encoding_dictionary do
            if code equals current_code then
                Append it to the output
                Reset bitarray to empty to append next bit
                Break out of loop
            end if
        end for
    end for
    Return output
```

III. Results

Tables II and III show that the algorithm that had the best performance was the Huffman Words. Although bzip2 had slightly better performance with a smaller file (higher compression ratio), the Huffman Words outperforms the other algorithms as the size of the file scales up. Its encoding and decoding time also outperforms the other algorithms.

Algorithm	Encode (s)	Decode (s)	Comp Ratio
Arithmetic	0.7796	0.8598	0.2038
Bzip2	0.0431	0.0457	1.7356
Huffman Words	0.0013	0.0002	1.2790
Huffman Letters	0.0011	0.0003	1.6458
Huffman top 10	0.0019	0.0003	1.6446
Huffman top 20	0.0018	0.0002	1.6358
Huffman top 30	0.0018	0.0002	1.6297

TABLE II. Short Sample (11037 Bytes)

Algorithm	Encode (s)	Decode (s)	Comp Ratio
Arithmetic	149.5693	247.4196	0.2039
bzip2	5.3654	8.8438	2.2722
Huffman Words	0.0781	0.0204	5.1094
Huffman Letters	0.1504	0.0502	1.8908
Huffman Top 10	0.3051	0.0504	1.9194
Huffman Top 20	0.2884	0.0479	1.9896
Huffman Top 30	0.2743	0.0456	2.0777

TABLE III. Larger Sample (2167737 Bytes)

IV. Conclusion

A. Analysis and Discussions

1) Arithmetic Coding: The implementation of the arithmetic coding algorithm for this project is not optimal. As can be seen from the results in Section III, this algorithm had the worst performance in both run time and compression ratio.

While modifications were applied to the infinite-precision approach of arithmetic coding (see Section II-A4), the main bottleneck in this algorithm is the size of the frequency tables. For the short text sample in Table II, the total file size of the encoded message (without the frequency tables) was approximately 5 KB. The original file size of the message was 11 KB. The compression ratio without the frequency tables would be closer to 2, which is a much better performance.

If, instead, the algorithm were to try and encode the entire message with no limit to the number of bits used in the encoded message, the algorithm would take hours to complete, because it is doing a lot of computations with high-precision floats.

So, the results of the algorithm's performance was to be expected since the finite-precision approach was not used. Even with modifications, like the ones implemented in this project, can only do so much to try and improve performance.

2) Huffman Coding: The performance characteristics exhibited by the various Huffman coding tests were about what was be expected before their testing.

The primary goal of using words instead of individual characters was to increase the compression ratio as inputs got larger, and that is shown by the testing. Because the implementation for characters and words is identical, it was also expected that the time taken would be roughly the same, as the algorithmic complexity for each remained the same. Interestingly, for a larger input encoding and decoding words was faster than it's letters counterpart. This is most likely due to less iterations occurring on a per-word basis compared to character by character iteration.

The goal of the second improvement, grouping by letters and most common words a success. The objective was to find a balancing point between words and letters such that the mapping file for symbols to bit codes wouldn't grow excessively large for smaller inputs, ruining any gains from the file itself being smaller. Looking at the data in Table II, the compression ratio for letters alone is greater than that of the hybrid or words only approach. However, only using the top 10 words is remarkably close to the ratio of letters along. Running the test again with the top 1-9 words in the file actually yielded a higher compression ratio for the top one, two, and three words. This shows that there is a turning point where the hybrid approach is relevant, but it required manual testing in order to find. In an ideal program, the expected size after compression could be calculated in advance for each of the implemented methods, and then the method with the smallest size chosen to actually compress the file.

3) bzip2: Overall, this was a successful project. Though different from the original implementation of bzip2, this compression algorithm is successful in compressing files. This implementation is not as effective at compressing text files as the original implementation of bzip2. However, for images, it is roughly equal. Neither the original compression nor this implementation compressed images very well, and in most cases slightly expanded the images. This was tested for jpg, png, tiff, and .NEF (.RAW) images.

For the text files, the original bzip2 compressed text files to roughly a 4:1 of the original size. In this implementation, it was on average compressing to at best 2:1, and at worst 1.2:1. In addition, this implementation took a bit longer than the original implementation. There are a few reasons for this.

For compute time, there are two major things that are drawbacks in this implementation. First, the entire Burrows-Wheeler matrix must be constructed for the encoder, which takes a lot of time and memory. One solution to this is implementing with suffix arrays. Second, some implementations use parallel processing for the blocks. This implementation is a for loop, so running the processes in parallel would greatly reduce the encode and decode runtime.

This implementation was worse at compression for a few reasons. The greatest reason for this is the limited block size in combination with the limited RLE2. The limited block size causes shorter strings of repeated characters. The RLE2 in this implementation can only handle 255 characters in a row. If the block size was increased to 900, and RLE2 could handle a string of 900k repeated characters, the compression ratios would be very similar. If the block size was increased, the resulting saved Huffman tables would also be much less than what they are currently. If a file of 18,000,000 bytes were split into blocks of 900kb, then only 2 Huffman tables at most would be required (likely only 1 table). In this implementation with 6kb blocks, 3000 Huffman tables are required to be saved which greatly hurts the compression ratio and saved file size.

Even though this implementation is not as successful as the original bzip2, it is still a valid compression software. With the improvements listed in the paper, this algorithm should be very close to, if not better than the original bzip2. The reason it would be better is because of the removal of the original RLE which can cause expansion.

B. Improvements

1) Arithmetic Coding: The biggest limitation of the arithmetic coding algorithm implemented in this project size of the frequency table list that accompanies the encoded message (see Section IV-A1). Since the message is encoded in chunks, a lot of same characters appear in multiple frequency tables. This makes the file where the frequency tables are saved much larger. This issue was slightly mitigated by compressing the frequency tables using bzip2, but the overall compression ratio is still approximately 1.

If only the encoded message was taken into account, the compression ratio would be closer to 2, but that is still very limited. The best way to implement arithmetic coding is by using finite-precision encoding and decoding. This approach uses integers instead of floats to generate the intervals of the number line. The finite-precision implementation would make it so that the entire message could be encoded, instead of being broken up into sub-strings. For an example of how finite-precision encoding works, see [3] and [11].

2) Huffman Coding: There are two primary improvements to be made to the Huffman Coding implementation.

The first is to use a better format for storing the mapping of symbols to bits. The current implementation stores the mapping in a JSON file, which means every code is octupled in size as each bit is stored as a 0 or 1 character. JSON is also interpreted as UTF-8, which restricts the implementation to only operating on a plain text file, and not on arbitrary files with bytes. Swapping to some sort of binary format would also provide the benefit of being able to embed the mapping directly into the Huffman encoded file. This could have been done with the current implementation, but then the output file would contain both plain text and encoded binary, and the decoding process would become much more complicated to implement.

The second improvement to make is to the format used to store the encoded file. The file is currently a stored in Python's pickle format [12], which is an efficient binary format, but one specific to Python. The purpose of the format is to save and load Python objects from disk, which presents a vector for arbitrary code execution. The primary reason for using pickle was to preserve the exact number of bits present in the bitarray object used for encoding the source data. When using the facilities of this object to write it's contents to disk it would result in padding being appended to the file, so pickle was utilized to simplify the decoding process. Switching to writing pure bytes would also allow for the mapping of symbols to bits to be embedded directly in the same file, preventing any possible mix up between mapping files and encoded data.

3) bzip2: The burrows wheeler could be implemented with a suffix array or FM index. This would greatly speed up the process and reduce memory.

The burrows wheeler decoding could also be improved with counting on the fly using an implementation of FM index as implemented in the paper [8].

In the original run length encoding, even though it is not implemented, there is a better way to get the counts using only 1 escape character. Instead of coding the length 4 times, it could only be coded once. Doing this, only 1 escape character would be necessary.

It would be good to create a separate, new file type containing all the necessary information for the decode instead of the 4 separate files as in the original bzip2 (.bz).

In the second RLE, it would be better to encode the way described in the paper to handle up to block_size_kb number of repeats [8].

REFERENCES

[1] [Online]. Available: https://go-compression.github.io/algorithms/arithmetic/

[2] A. Gad, "Lossless data compression using arithmetic encoding in python and its applications in deep learning," Aug 2023. [Online]. Available: https://neptune.ai/blog/lossless-data-compression-using-arithmetic-encoding-in-python-and-its-applications-in-deep-learning

[3] M. Nelson, "Data compression with arithmetic coding," Oct 2014. [Online]. Available: https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html

[4] [Online]. Available: https://docs.python.org/3/library/decimal.html#decimal.MAX_PREC

[5] "7.20. proof of optimality for huffman coding — cs3 data structures algorithms." [Online]. Available: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/HuffProof.html

[6] B. Bird, "Data compression (summer 2020) - lecture 13 - bzip2," Marmonth=Jun 20217. [Online]. Available: https://www.youtube.com/watch?v=9aAxBvbhl4k&t=458s

[7] J. Julian Seward, "bzip2 and libbzip2, version 1.0.8." [Online]. Available: https://sourceware.org/bzip2/manual/manual.html

[8] D. J. Burrows, M.; Wheeler, "A block-sorting lossless data compression algorithm," https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html, [Accessed 18-12-2023].

[9] I. Schnell, "ilanschnell/bitarray," Nov 2008. [Online]. Available: https://github.com/ilanschnell/bitarray

[10] "heapq — heap queue algorithm — python 3.12.1 documentation." [Online]. Available: https://docs.python.org/3/library/heapq.html

[11] "(IC 5.13) Finite-precision arithmetic coding - Encoder — youtube.com," https://www.youtube.com/watch?v=9vhbKiwjJo8&list=PLE125425EC837021F&index=53&t=677s, [Accessed 19-12-2023].

[12] "pickle — python object serialization — python 3.12.1 documentation." [Online]. Available: https://docs.python.org/3/library/pickle.html