Algorithm Engineering

CPSC 335-12-21575

Dr. Syed Hassan Shah

Huffman Coding Compression Tool Project Report

Group #9

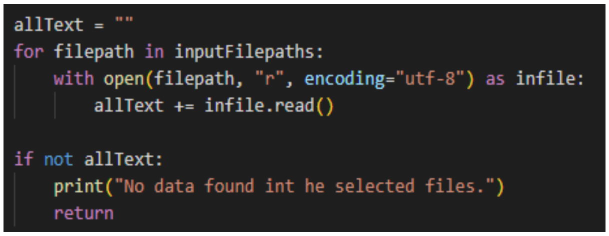
Developer 1: Chris Manlove

Developer 2: Jaxon Mcconnell

Developer 3: Donovan Banks

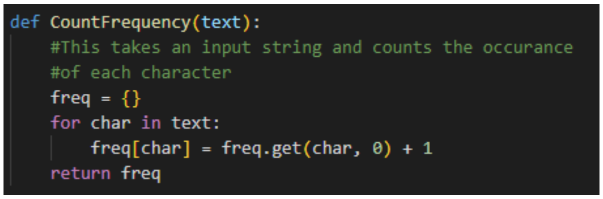
A critical component of modern computing is the ability to compress files, many serices we interact with each day benefit from this technology. Compression video and audio enabled video streaming services like Netflix and Youtube to operate at lower costs while simultaneously improving user experience by adjusting compression to match the bandwidth capacity of clients. Given the nature of the data these services handle, a variable amount of loss is often tolerable as it will only be noticeable via specific audio/visual artifacts. Such loss is often not tolerable for companies that host email, archival, and file sharing services and as such there are also lossless compression algorithms that can be used to meet the specific needs of the service. An example of such a lossless compression algorithm is huffman coding, which optimizes single symbol based text, such as this paper, by overwriting the fixed-length ascii table with a custom variable length table that ensures more frequently used symbols are represented with smaller binary values. Our project will use huffman coding to implement a lossless .txt file compressor, and the critical algorithm steps will be outlined below.

<table>
<tr>
<td>First we read the data into working memory. It should be noted that for this specific program we assume the file will be small enough to fit into working memory from a single read.</td>
<td>



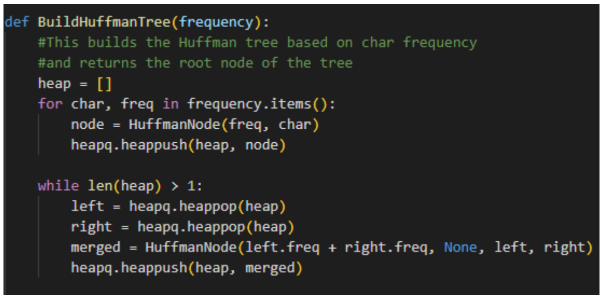
```
allText = ""
for filepath in inputFilepaths:
    with open(filepath, "r", encoding="utf-8") as infile:
        allText += infile.read()

if not allText:
    print("No data found int he selected files.")
    return
```

</td>
</tr>
<tr>
<td>Second we identify every unique character and count the frequencies</td>
<td>



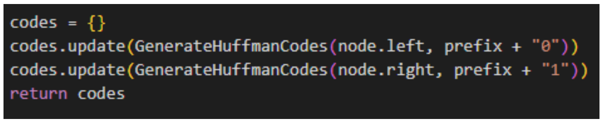
```
def CountFrequency(text):
    #This takes an input string and counts the occurance
    #of each character
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    return freq
```

</td>
</tr>
<tr>
<td>Third we use the min-heap data structure as a model to build the huffman tree for this specific data</td>
<td>



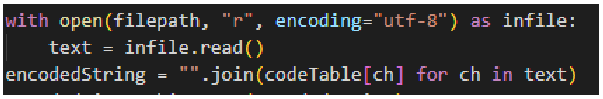
```
def BuildHuffmanTree(frequency):
    #This builds the Huffman tree based on char frequency
    #and returns the root node of the tree
    heap = []
    for char, freq in frequency.items():
        node = HuffmanNode(freq, char)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(left.freq + right.freq, None, left, right)
        heapq.heappush(heap, merged)
```

</td>
</tr>
<tr>
<td>Fourth we traverse the tree from the root to each leaf using 0 to track left edge traversals and 1 to track right edge traversals, using this to generate a dictionary of each symbol and its respective huffman code</td>
<td>



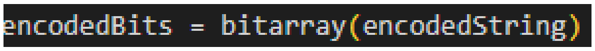
```
codes = {}
codes.update(GenerateHuffmanCodes(node.left, prefix + "0"))
codes.update(GenerateHuffmanCodes(node.right, prefix + "1"))
return codes
```

</td>
</tr>
<tr>
<td>Fifth we use the dictionary to substitute each symbol for the specific huffman code computed.</td>
<td>

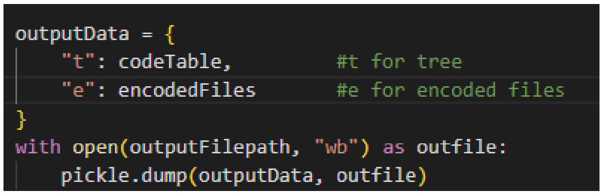


```
with open(filepath, "r", encoding="utf-8") as infile:
    text = infile.read()
encodedString = "".join(codeTable[ch] for ch in text)
```

</td>
</tr>
<tr>
<td>Lastly we write the binary data to a separate file. We use the bitarray library to convert a string of 0s and 1s into an actual bit array which we can then write directly to a binary file.</td>
<td>



```
encodedBits = bitarray(encodedString)
```

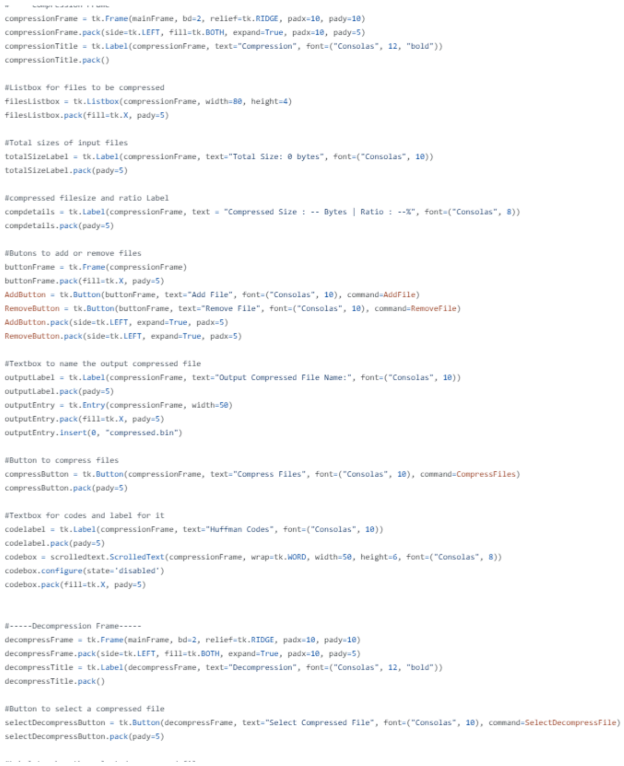


```
outputData = {
    "t": codeTable,         #t for tree
    "e": encodedFiles       #e for encoded files
}
with open(outputFilepath, "wb") as outfile:
    pickle.dump(outputData, outfile)
```

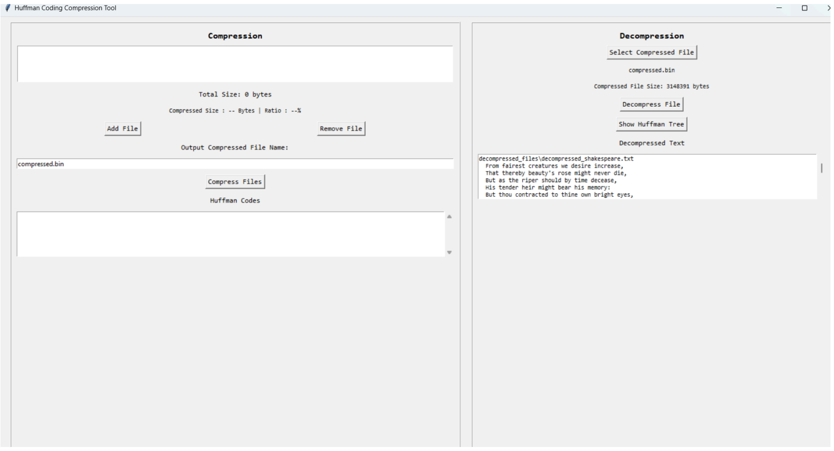
</td>
</tr>
</table>

The compression algorithm and file handling are critical under the hood components for the application that we were asked to design, development UI was essentially a large amount of busy work in designing and spacing out every element and ensuring that the buttons would actually execute the core algorithm with the correct inputs provided.

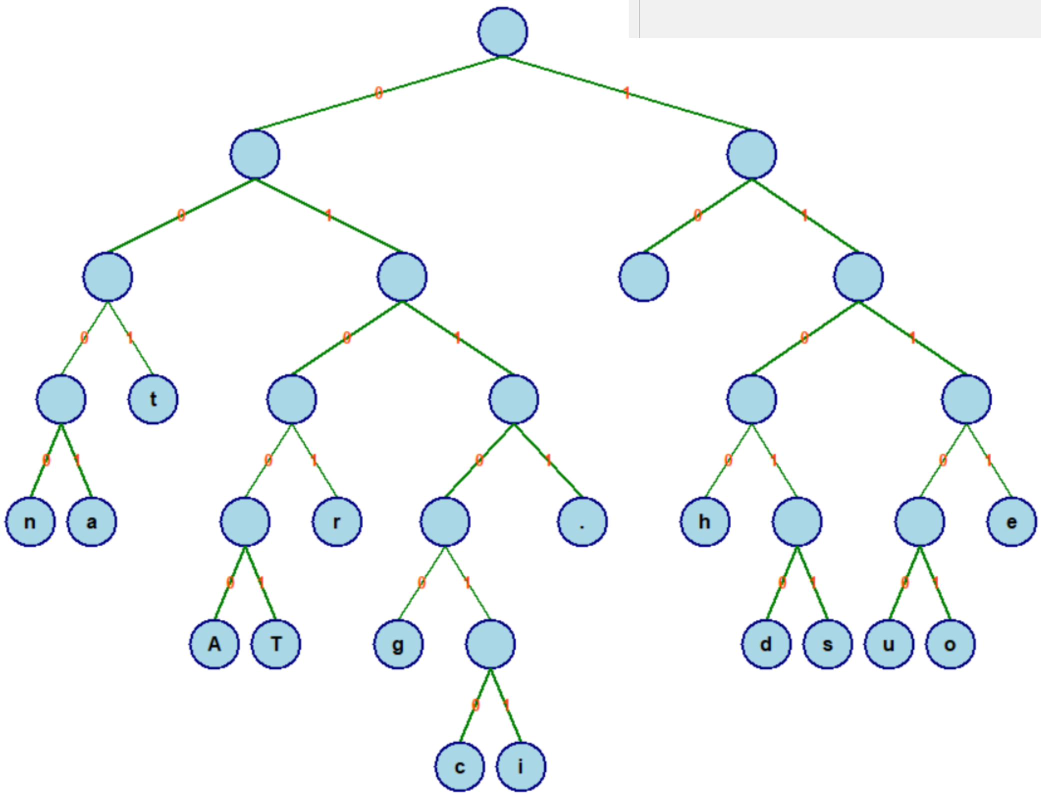
As you can see from the picture on the left, a majority of the code in the ui is simply adjusting ui element properties. We expended a significant amount of resources rendering a graph of the huffman tree on a canvas as can be seen below. Along with an image of the core application ui.



```
compressionFrame = tk.Frame(mainFrame, bd=2, relief=tk.RIDGE, padx=10, pady=10)
compressionFrame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=10, pady=5)
compressionTitle = tk.Label(compressionFrame, text="Compression", font=("Consolas", 12, "bold"))
compressionTitle.pack()

#Listbox for files to be compressed
filesListbox = tk.Listbox(compressionFrame, width=80, height=4)
filesListbox.pack(fill=tk.X, pady=5)

#Total sizes of input files
totalSizeLabel = tk.Label(compressionFrame, text="Total Size: 0 bytes", font=("Consolas", 10))
totalSizeLabel.pack(pady=5)

#compressed filesize and ratio Label
compdetails = tk.Label(compressionFrame, text = "Compressed Size : -- Bytes | Ratio : --%", font=("Consolas", 8))
compdetails.pack(pady=5)

#Butons to add or remove files
buttonFrame = tk.Frame(compressionFrame)
buttonFrame.pack(fill=tk.X, pady=5)
AddButton = tk.Button(buttonFrame, text="Add File", font=("Consolas", 10), command=AddFile)
RemoveButton = tk.Button(buttonFrame, text="Remove File", font=("Consolas", 10), command=RemoveFile)
AddButton.pack(side=tk.LEFT, expand=True, padx=5)
RemoveButton.pack(side=tk.LEFT, expand=True, padx=5)

#Textbox to name the output compressed file
outputLabel = tk.Label(compressionFrame, text="Output Compressed File Name:", font=("Consolas", 10))
outputLabel.pack(pady=5)
outputEntry = tk.Entry(compressionFrame, width=50)
outputEntry.pack(fill=tk.X, pady=5)
outputEntry.insert(0, "compressed.bin")

#Button to compress files
compressButton = tk.Button(compressionFrame, text="Compress Files", font=("Consolas", 10), command=CompressFiles)
compressButton.pack(pady=5)

#Textbox for codes and label for it
codelabel = tk.Label(compressionFrame, text="Huffman Codes", font=("Consolas", 10))
codelabel.pack(pady=5)
codebox = scrolledtext.ScrolledText(compressionFrame, wrap=tk.WORD, width=50, height=6, font=("Consolas", 8))
codebox.configure(state='disabled')
codebox.pack(fill=tk.X, pady=5)


#-----Decompression Frame-----
decompressFrame = tk.Frame(mainFrame, bd=2, relief=tk.RIDGE, padx=10, pady=10)
decompressFrame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=10, pady=5)
decompressTitle = tk.Label(decompressFrame, text="Decompression", font=("Consolas", 12, "bold"))
decompressTitle.pack()

#Button to select a compressed file
selectDecompressButton = tk.Button(decompressFrame, text="Select Compressed File", font=("Consolas", 10), command=SelectDecompressFile)
selectDecompressButton.pack(pady=5)
```



```
decompressed_files\decompressed_shakespeare.txt
  From fairest creatures we desire increase,
  That thereby beauty's rose might never die,
  But as the riper should by time decease,
  His tender heir might bear his memory:
  But thou contracted to thine own bright eyes,
```


0
1
t
n
a
r
.
h
e
A
T
g
d
s
u
o
c
i