**Binary File Compression: Huffman Encoding**

By Vasavi Vempati

11/21/2019

**Table of Contents**

1. Introduction 2

1.1. Document Outline 3

1.2. Document Description 5

1.2.1. Introduction 5

1.2.2. System Overview 5

2. Design Considerations 5

2.1. Assumptions and Dependencies 6

2.2. General Constraints 6

2.3. Goals and Guidelines 6

2.4. Development Methods 7

3. Architectural Strategies 7

4. System Architecture 7

4.1. Subsystem Architecture 8

5. Policies and Tactics 9

6. Detailed System Design 10

6.1. Classification 10

6.2. Definition 10

6.3. Responsibilities 10

6.4. Constraints 10

6.5. Composition 10

6.6. Uses/Interactions 10

6.7. Resources 11

6.8. Processing 11

6.9. Interface/Exports 11

6.10. Detailed Subsystem Design 11

7. Glossary 12

8. Bibliography 12

# Introduction

The following is a detailed description of all the technologies used in order to create my binary file compression web application. The language I chose to write the application in is Python 3.8. I chose this language because I liked some of the comparison and iteration techniques available in this version of python. I also chose to use the python web micro-framework Flask because it is very lightweight, easy to set up, and easy to test with. In addition to Flask, I used Flask-Bootstrap for UI manipulation and design. Finally, I used Anaconda to create a Conda environment to manage all my dependencies so that the libraries I used would not conflict with those installed in my main development environment. I chose not to deploy to web-server because I believe my application is simple enough that it will be fairly easy to run locally.

## Document Description

Below I will describe the approach I took to solving this file compression problem, the technologies I used, and the significance of the output produced by my program in relation to the sample data provided. I will also describe how to set up and run my application locally after cloning the repository from GitHub (<https://github.com/vvempati/huffmanEncoding>) .

### Introduction

* The purpose of this document is to explain my approach to file compression via Huffman encoding.
* The scope of this document will cover launching the application and explaining the code written.
* This document is intended for anyone who wants to learn about Huffman encoding and why it is a great lossless compression technique.
* I am using Python 3.8, Flask 1.1.1, Flask-Bootstrap 3.3.7.1
* I used this guide: <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>

### System Overview

This software system consists of a Python web application deployed using the python web micro-framework: Flask. The application consists of a folder called ‘app’ which contains all the pertinent classes and files. The ‘app’ folder consists of three classes ‘\_\_init\_\_.py’, ‘routes.py’, and ‘huffman.py’. The first class ‘\_\_init\_\_.py’ imports all the required dependencies and creates instances of Flask and Bootstrap that are needed for the rest of the application. The second class ‘routes.py’, describes the three different URI’s (uniform resource identifies) related to the homepage (‘/index’), the compression page (‘/compress-data’), and the final output page(‘/return-files’). The third class ‘huffman.py’ details the file parsing, creation of the Huffman binary tree in the form of a min-heap and finally compressing the file input into a smaller size using the Huffman binary tree. The ‘app’ folder consists of three sub-folder called ‘outputs’, ‘templates’, uploads’. The ‘outputs’ folder contains the output file from the compression algorithm. The ‘templates’ folder contains the HTML UI templates for displaying the data on the index, compression, and output pages. The ‘uploads’ folder contains the input binary files uploaded by the user for compression.

# Design Considerations

## Assumptions and Dependencies

Assumptions:

* + All data is uploaded in a binary file format with Signed Int 24 values in Big Endian format
  + Data input is compressed by assuming that each Signed Int 24 is a value unto its own and compression is commenced by looking for repetitions of those values
  + End-user will be able to download appropriate versions of Python, Bootstrap, and Bootstrap-Flask

## Approach

* Once the user uploads the file, the values are read 3 bytes at a time and placed into a dictionary. The key for this dictionary is the value read in and the value for that key is the frequency at which the key occurs in the file.
* A Node class is created with a pointer to the left and right child Nodes, a value attribute, and a frequency attribute. The Node class also has a custom comparison function that allows the objects of Node type to be sorted by frequency of value occurrence in a given collection.
* In order to create a min-heap the dictionary needs a collection with the reverse of this model: [value, frequency].
* A list of lists is then created from the dictionary and then for each of those lists tuple is created
* From that list of tuples, a heap can be created using Python’s heapq library
* Given the min-heap of tuples, a min-heap Node objects can be created
* With that min-heap of Node objects the min-heap is reorganized to create the Huffman tree by extracting two Nodes with the minimum frequency from the min-heap, adding together their frequencies of occurrences, creating a new internal node with the added frequency as its value and the first extracted node as its left child and the second extracted node as its right child, and adding this internal node back to the min heap.
* With the above step, a binary tree is created where the nodes representing the values with the most frequency of occurrence in the input are closest to the root node. All nodes in the binary tree that have an actual input value associated with them are leaf nodes.
* With the created binary tree, the compression codes can now be calculated by assigning a “0” for every left-child traversal and a “1” for every right-child traversal.
* The output file is then created by replacing the original values with the codes created for them.
* Finally, the compression ratio is calculated by
  + Taking the sum of all the products of character frequency and Huffman code length.
  + In order compare the byte value of the output file and the input file, we divide the above sum by 8, because that sum represents the number of bits taken up by the output file
  + Then the ratio is calculated by: Output file size(in bytes)/ Input file size(in bytes)

## Setup and Deployment

* In order to run this application locally
  + In terminal or command prompt run
    - git clone https://github.com/vvempati/huffmanEncoding
  + Then after moving into the **huffmanEncoding** directory
    - Download the latest installer of Python3.8 from python.org/downloads
    - Make sure the command python3 returns the correct version of python
    - If pip does not exist on the development machine being used to run this application, it can be installed by running the commands
      * curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
      * python get-pip.py
    - Then install Flask by running
      * pip install flask
    - Then install Flask-Bootstrap by running
      * pip install flask-bootstrap
* Now the development environment should be good to go, so the user may use terminal or the command prompt to go to the directory where the application was downloaded to and run the command:
  + flask run
* The application should be up and running on a local port at URL such as
  + http://127.0.0.1:5000

## Conclusion & Explanation of Compression Algorithm Choice

* I chose the Huffman compression algorithm because
  + It is used to convert fixed length codes into variable length codes and the given input values were of the same length
  + It is viable for any data type, so I knew it would be easy to apply to ECG data
  + It has minimum redundancy, which allows the average bytes required to represent a value to be as small as possible. I knew this would be important given that typically ECG Data usually has lots of repeating data in the flat portion of the waveform describing that data.

In conclusion, this project was very fun for me because it allowed me to learn a lot about how Python handles large sets of data and the importance of compression especially as it relates to healthcare data.