參考資料:

Huffman Coding Python Implementation: <https://bhrigu.me/post/huffman-coding-python-implementation/>

Huffman coding: <https://en.wikipedia.org/wiki/Huffman_coding>

繳交的python code參考至Huffman Coding Python Implementation，內含有個人註解

!! 點選UseHuffmanCoding.py可以將本文件壓縮成bin檔案，然後解壓縮成REAdME\_decompressed.txt

**說明:**

**Huffman coding** 是無損數據壓縮技術之一，常處理編寫工作。它根據輸入字符的出現頻率為輸入字符分配可變長度代碼。如果字符出現的頻率越高，則給字符的碼越短，相反則字符號碼越長。

* 資料壓縮

假設我們要給一個英文單字**"F O R G E T"**進行霍夫曼編碼，而每個英文字母出現的頻率分別如下

Symbol F O R G E T

Frequency 2 3 4 4 5 7

CODE 000 001 100 101 011 111

接下來就是找最小節點相加!

step1: 2+3 = 5 (5, 4, 4, 5, 7)

step2: 4+4 = 8 (5, 8, 5, 7)

step3: 5+5 = 10 (10, 8, 7)

step4: 8+7 = 15 (10, 15)

final: 10+15 = 25 (25) --> 只剩一個，結束

最後的樹就會長的像這樣:

25

10 15

5 8

2 3 5 4 4 7

F O E R G T

* 資料解壓縮

霍夫曼碼樹的解壓縮就是將得到的前置碼（Prefix Huffman code; 就是0和1(判斷是左子節點還是右子節點)）轉換回符號，通常藉由樹的追蹤（Traversal），將接收到的位元串（Bits stream）一步一步還原。但是要追蹤樹之前，必須要先重建霍夫曼樹；某些情況下，如果每個符號的權重可以被事先預測，那麼霍夫曼樹就可以預先重建，並且儲存並重複使用，否則，傳送端必須預先傳送霍夫曼樹的相關資訊給接收端。

最簡單的方式，就是預先統計各符號的權重並加入至壓縮之位元串，但是此法的運算量花費相當大，並不適合實際的應用。若是使用Canonical encoding，則可精準得知樹重建的資料量只占*B*2^*B*位元（其中B為每個符號的位元數（bits））。如果簡單將接收到的位元串一個位元一個位元的重建，例如：'0'表示父節點，'1'表示終端節點，若每次讀取到1時，下8個位元則會被解讀是終端節點（假設資料為8-bit字母），則霍夫曼樹則可被重建，以此方法，資料量的大小可能為2~320位元組不等。雖然還有很多方法可以重建霍夫曼樹，但因為壓縮的資料串包含"trailing bits"，所以還原時一定要考慮何時停止，不要還原到錯誤的值，如在資料壓縮時時加上每筆資料的長度等

**討論:**

Q: 為甚麼可以減少資料量?

A: '0'與'1'分別代表指向左子節點與右子節點，最後為完成的二元樹共有**n**個終端節點(一開始的元素)點與**n-1**個非終端節點，去除了不必要的符號並產生最佳的編碼長度，先把樹狀結構轉乘01組成的值，再把它轉成byte組成的資料變成 bin檔，資料量就可以少很多了。

Q: 資料要如何還原?

透過樹去反推，先用bin檔還原成 01的檔案(Prefix Huffman code)，再把它還原成字符，就可以完整的還原資料了

Python 主要程式碼解析:

def compress(self):  
  """  
      主要呼叫函數，處理檔案的壓縮  
      :return: 各個字符出現次數  
  """  
  filename, file\_extension = os.path.splitext(self.path) # 取得輸入的文件  
  output\_path = filename + ".bin" # 輸出的壓縮檔名稱  
​  
  with open(self.path, 'r+') as file, open(output\_path, 'wb') as output:  
      text = file.read()  
      text = text.rstrip()  
​  
      frequency = self.make\_frequency\_dict(text) # 創建字典  
      self.make\_heap(frequency) # 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的最下層(終端節點)  
      self.merge\_nodes() # 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的其他層(非終端節點)  
      self.make\_codes() # 把相連接的資料用 0、1 表示他是 左子節點 還是 右子節點  
​  
      encoded\_text = self.get\_encoded\_text(text) # 把壓縮好(self.codes[char])的資料傳回  
      padded\_encoded\_text = self.pad\_encoded\_text(encoded\_text) # 進行編碼  
​  
      b = self.get\_byte\_array(padded\_encoded\_text) # 取得byte編碼內容  
      output.write(bytes(b)) # 寫入檔案 .bin ???.bin  
​  
  print("Compressed")  
  return output\_path

def decompress(self, input\_path):  
  """  
      主要呼叫函數，處理檔案的解壓縮  
      :param input\_path: .txt檔傳入的文字內容  
      :return: 輸出檔案，還原原本的資料  
  """  
  filename, file\_extension = os.path.splitext(self.path)  
  output\_path = filename + "\_decompressed" + ".txt"  
# 輸出文字檔   ???\_decompressed\_text  
  with open(input\_path, 'rb') as file, open(output\_path, 'w') as output:  
      bit\_string = ""  
​  
      byte = file.read(1)  
      while (len(byte) > 0): # 讀取文檔  
          byte = ord(byte)  
          bits = bin(byte)[2:].rjust(8, '0')  
          bit\_string += bits  
          byte = file.read(1)  
​  
      encoded\_text = self.remove\_padding(bit\_string) # 把文件byte 轉成 Prefix Huffman code  
​  
      decompressed\_text = self.decode\_text(encoded\_text) # 把 0 1 編碼轉乘 文字出現次數 的 值， 利用樹狀結構去回推 (reverse\_mapping)  
​  
      output.write(decompressed\_text) # 寫入文字檔   ???\_decompressed\_text  
​  
  print("Decompressed")  
  return output\_path

完整python程式碼:

UseHuffmanCoding.py

from HuffmanCoding import HuffmanCoding  
import sys  
  
path = "README.txt" # 這邊可以用我的介紹檔案做測試  
  
h = HuffmanCoding(path)  
  
output\_path = h.compress() # 先做壓縮  
print("Compressed file path: " + output\_path)  
  
decom\_path = h.decompress(output\_path) # 再做解壓縮  
print("Decompressed file path: " + decom\_path)

HuffmanCoding.py

import heapq  
import os  
  
"""  
author: Bhrigu Srivastava  
website: https:bhrigu.me  
chinese annotation: 110910541  
"""  
  
  
# 這裡出現的目的是讓下面class HuffmanCoding 裡面 class HeapNode 的if (not isinstance(other, HeapNode)):可以正常運作  
class HeapNode:  
 def \_\_init\_\_(self, char, freq):  
 self.char = char  
 self.freq = freq  
 self.left = None  
 self.right = None  
  
 # defining comparators less\_than and equals  
 def \_\_lt\_\_(self, other):  
 return self.freq < other.freq  
  
 def \_\_eq\_\_(self, other):  
 if (other == None):  
 return False  
 if (not isinstance(other, HeapNode)):  
 return False  
 return self.freq == other.freq  
  
  
class HuffmanCoding: # 可以創建霍夫曼碼  
 def \_\_init\_\_(self, path):  
 self.path = path # 接收文件路徑  
 self.heap = [] # 使用在 heapq.heappush(self.heap, node)，接收最下層的終端節點  
 self.codes = {} # 霍夫曼碼樹 dict[char] = 0 || 1， 0代表是左子節點，1代表是右子節點  
 self.reverse\_mapping = {}  
  
 class HeapNode: # 霍夫曼碼樹節點連接  
 def \_\_init\_\_(self, char, freq):  
 self.char = char # 儲存字元  
 self.freq = freq # 儲存字元出現次數  
 self.left = None # 樹狀結構紀錄  
 self.right = None  
  
 # defining comparators less\_than and equals  
 def \_\_lt\_\_(self, other):  
 return self.freq < other.freq  
  
 def \_\_eq\_\_(self, other):  
 if (other == None):  
 return False  
 if (not isinstance(other, HeapNode)):  
 return False  
 return self.freq == other.freq  
  
 # functions for compression:  
  
 def make\_frequency\_dict(self, text):  
 *"""  
 統計各個字符在檔案中出現的次數  
 :param text: .txt檔傳入的文字內容  
 return: 各個字符出現次數  
 """* frequency = {} # 創建字典，主要是存取text裡面的字出現的次數  
 for character in text:  
 if not character in frequency:  
 frequency[character] = 0 # 一開始創建，初始值為0  
 frequency[character] += 1 # 每重複一次就加一  
 return frequency  
  
 def make\_heap(self, frequency):  
 *"""  
 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的最下層(終端節點)  
 :param frequency: 各個字符出現次數  
 """* for key in frequency:  
 node = self.HeapNode(key, frequency[key])  
 heapq.heappush(self.heap, node) # 把資料丟入霍夫曼碼樹  
  
 def merge\_nodes(self):  
 *"""  
 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的其他層(非終端節點)  
 """* while (len(self.heap) > 1):  
 node1 = heapq.heappop(self.heap)  
 node2 = heapq.heappop(self.heap)  
  
 merged = self.HeapNode(None, node1.freq + node2.freq) # 把最小的兩個終端節點相加  
 merged.left = node1 # 節點相連  
 merged.right = node2  
  
 heapq.heappush(self.heap, merged) # 把資料丟入霍夫曼碼樹  
  
 def make\_codes\_helper(self, root, current\_code):  
 *"""  
 把相連接的資料用 0、1 表示他是 左子節點 還是 右子節點  
 :param root: .txt檔傳入的文字內容  
 :param current\_code: .txt檔傳入的文字內容  
 :return: 在沒有資料或是在root.char都已經填過值十，回傳null，  
 """* if (root == None):  
 return  
  
 if (root.char != None): # 把每個相連接的地方分別填入0、1  
 self.codes[root.char] = current\_code  
 self.reverse\_mapping[current\_code] = root.char  
 return  
  
 self.make\_codes\_helper(root.left, current\_code + "0")  
 self.make\_codes\_helper(root.right, current\_code + "1")  
  
 def make\_codes(self):  
 *"""  
 使用 self.make\_codes\_helper  
 """* root = heapq.heappop(self.heap) # 取得最下層的內容  
 current\_code = ""  
 self.make\_codes\_helper(root, current\_code)  
  
 def get\_encoded\_text(self, text):  
 *"""  
 把壓縮好(self.codes[char])的資料傳回  
 :param text: .txt檔傳入的文字內容  
 :return: self.codes裡面的內容  
 """* encoded\_text = ""  
 for character in text:  
 encoded\_text += self.codes[character] # 把上面 make\_codes 填入的那些 0 1傳入到 encoded\_text  
 return encoded\_text  
  
 def pad\_encoded\_text(self, encoded\_text):  
 *"""  
 進行編碼  
 :param encoded\_text: self.codes裡面的內容  
 :return: 轉成特殊編碼  
 """* extra\_padding = 8 - len(encoded\_text) % 8  
 for i in range(extra\_padding):  
 encoded\_text += "0"  
  
 padded\_info = "{0:08b}".format(extra\_padding) # 把char轉成byte  
 encoded\_text = padded\_info + encoded\_text  
 return encoded\_text  
  
 def get\_byte\_array(self, padded\_encoded\_text):  
 *"""  
 取得byte編碼內容  
 :param padded\_encoded\_text: 特殊編碼  
 :return: byte編碼  
 """* if (len(padded\_encoded\_text) % 8 != 0):  
 print("Encoded text not padded properly")  
 exit(0)  
  
 b = bytearray()  
 for i in range(0, len(padded\_encoded\_text), 8):  
 byte = padded\_encoded\_text[i:i + 8]  
 b.append(int(byte, 2))  
 return b  
  
 def compress(self):  
 *"""  
 主要呼叫函數，處理檔案的壓縮  
 :return: 各個字符出現次數  
 """* filename, file\_extension = os.path.splitext(self.path) # 取得輸入的文件  
 output\_path = filename + ".bin" # 輸出的壓縮檔名稱  
  
 with open(self.path, 'r+') as file, open(output\_path, 'wb') as output:  
 text = file.read()  
 text = text.rstrip()  
  
 frequency = self.make\_frequency\_dict(text) # 創建字典  
 self.make\_heap(frequency) # 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的最下層(終端節點)  
 self.merge\_nodes() # 利用 heapq.heappush、 self.heap、 class HeapNode 製作 霍夫曼碼樹 的其他層(非終端節點)  
 self.make\_codes() # 把相連接的資料用 0、1 表示他是 左子節點 還是 右子節點  
  
 encoded\_text = self.get\_encoded\_text(text) # 把壓縮好(self.codes[char])的資料傳回  
 padded\_encoded\_text = self.pad\_encoded\_text(encoded\_text) # 進行編碼  
  
 b = self.get\_byte\_array(padded\_encoded\_text) # 取得byte編碼內容  
 output.write(bytes(b)) # 寫入檔案 .bin ???.bin  
  
 print("Compressed")  
 return output\_path  
  
 """ functions for decompression: """  
  
 def remove\_padding(self, padded\_encoded\_text):  
 *"""  
 把文件byte 轉成 Prefix Huffman code  
 :param padded\_encoded\_text: byte code  
 :return: 把 byte 轉成 正常編碼  
 """* padded\_info = padded\_encoded\_text[:8]  
 extra\_padding = int(padded\_info, 2)  
  
 padded\_encoded\_text = padded\_encoded\_text[8:]  
 encoded\_text = padded\_encoded\_text[:-1 \* extra\_padding]  
  
 return encoded\_text  
  
 def decode\_text(self, encoded\_text):  
 *"""  
 把 0 1 編碼轉乘 文字出現次數 的 值， 利用樹狀結構去回推 (reverse\_mapping)  
 :param encoded\_text: 正常編碼  
 :return: 壓縮文檔文字  
 """* current\_code = ""  
 decoded\_text = ""  
  
 for bit in encoded\_text:  
 current\_code += bit  
 if (current\_code in self.reverse\_mapping):  
 character = self.reverse\_mapping[current\_code]  
 decoded\_text += character  
 current\_code = ""  
  
 return decoded\_text  
  
 def decompress(self, input\_path):  
 *"""  
 主要呼叫函數，處理檔案的解壓縮  
 :param input\_path: .txt檔傳入的文字內容  
 :return: 輸出檔案，還原原本的資料  
 """* filename, file\_extension = os.path.splitext(self.path)  
 output\_path = filename + "\_decompressed" + ".txt"  
  
 with open(input\_path, 'rb') as file, open(output\_path, 'w') as output: # open 輸出文字檔 ???\_decompressed\_text  
 bit\_string = ""  
  
 byte = file.read(1)  
 while (len(byte) > 0): # 讀取文檔  
 byte = ord(byte)  
 bits = bin(byte)[2:].rjust(8, '0')  
 bit\_string += bits  
 byte = file.read(1)  
  
 encoded\_text = self.remove\_padding(bit\_string) # 把文件byte 轉成 Prefix Huffman code  
  
 decompressed\_text = self.decode\_text(encoded\_text) # 把 0 1 編碼轉乘 文字出現次數 的 值， 利用樹狀結構去回推 (reverse\_mapping)  
  
 output.write(decompressed\_text) # 寫入文字檔  
  
 print("Decompressed")  
 return output\_path