Telekomunikacja - laboratorium	Studia dzienne - inżynierskie

Nazwa zadania	Algorytm statycznego kodowania Huffmana				
Dzień	poniedziałek	**Godzina**	12:15 – 13:45	**Rok akademicki**	2019/2020

Imię i Nazwisko	Konrad Chojnacki 224274
Imię i Nazwisko	Krzysztof Szcześniak 224434
Imię i Nazwisko	----------

Opis programu, rozwiązania problemu.

1. Program odczytuje łańcuch znaków z podanego pliku tekstowego.
2. Następnie mamy możliwość zakodować odczytany łańcuch znaków za pomocą algorytmu kodowania Huffmana, który składa się z następujących etapów:
   a) pierwszym etapem jest sporządzenie słownika częstotliwości występowania dla każdego znaku w podanym tekście
   b) mając taki słownik można utworzyć listę drzew binarnych przechowujących pary: znak i jego częstość występowania
   c) kolejnym etapem jest zbudowanie drzewa Huffmana, które polega na:
      - bierzemy z listy dwa drzewa, które mają najmniejszą częstotliwość
      - łączymy dwa wybrane wcześniej drzewa w jedno, które nie przechowuje znaku, ale przechowuje sumę częstotliwości występowania tych znaków w tekście oraz przechowuje wskazania do tych dwóch drzew (drzewu, które miało mniejszą częstotliwość odpowiada wskazanie lewe a większemu prawe)
      - wykonujemy te dwa podpunkty dopóki na liście będzie więcej niż jedno drzewo
   d) mając drzewo kodowe Huffmana przechodzimy do etapu tworzenia słownika kodowego, który polega na:
      - każdej lewej krawędzi drzewa przypisujemy umownie 0, a prawej 1
      - przechodzimy drzewo od korzenia do każdego liścia: jeśli liść znajduje się po prawej stronie dopisywany jest bit o wartości 1 do kodu, a dla liścia znajdującego się po lewej dopisywany jest bit o wartości 0
      - tym sposobem tworzymy słownik kodowy przechowujący pary: znak i zakodowana postać znaku, której długość jest równa głębokości znaku w drzewie
   e) kodujemy tekst odczytany z podpunktu 1 za pomocą sporządzonego słownika
   f) zauważmy, że otrzymany kod ma postać binarną, a więc zamieniamy go na bajty i zapisujemy do pliku. Tutaj występuje pewien problem, a mianowicie jeśli liczba znaków w tekście nie jest podzielna przez 8 to musimy dopełnić kod bitami 0. Dokonuje się kompresja pliku, ponieważ zawiera on mniej znaków niż plik początkowy tym sposobem też mniej zajmuje miejsca na dysku.
3. Zakodowany plik, informacje o dodanych bitach oraz słownik przesyłamy za pomocą gniazd sieciowych do drugiej instancji programu którego zadaniem jest odkodować zakodowany plik- dokonać dekompresji. W tym celu:
   a) odczytujemy bajty z zakodowanego pliku oraz zamieniamy je na postać binarną oraz usuwamy z końca przekazaną ilość bitów
   b) oprócz pliku został przesłany także słownik, a więc zamieniamy odczytaną postać binarną za pośrednictwem słownika na właściwy tekst
   c) odczytany tekst zapisujemy do pliku – powinien być taki sam jak plik z podpunktu 1

Najważniejsze elementy kodu programu z opisem.

1. Stworzenie słownika częstotliwości występowania każdego znaku w podanym tekście. Stworzenie słownika w języku python w którym występujące znaki są kluczami, a wartościami jest liczba wystąpień znaku. Liczba ta jest obliczana w pętli przechodzącej przez cały tekst i jeśli dany znak występuje w tekście liczba jest inkrementowana.

```
def calculateFrequencies(self, pathToText):
    file = open(pathToText, 'r')
self.text = file.read()
    file.close()
for letter in self.text:
if letter in self.frequencies:
self.frequencies[letter] += 1
else:
self.frequencies[letter] = 1
```

2. Tworzenie drzewa Huffmana – na początek tworzymy listę drzew binarnych wykorzystując zaimplementowaną klasę TreeNode, a następnie pętla while realizuje budowanie drzewa – opisane w pierwszej sekcji sprawozdania. Aby wybrać dwa drzewa o najmniejszej częstotliwości stosujemy kolejkę priorytetową heapq. Polecenie heapq.heappop zwraca najmniejsze drzewo które aktualnie jest na liście. Konieczne było zaimplementowanie w klasie TreeNode odpowiedniego

komparatora.

```
def buildHuffmanTree(self):
   [heapq.heappush(self.heapNodes, TreeNode(letter, self.frequencies[letter])) for letter in self.frequencies]

while self.heapNodes.__len__() >1:
      smallestNodes = [heapq.heappop(self.heapNodes) for i in range(2)]
      heapq.heappush(self.heapNodes, TreeNode(None, smallestNodes[0].frequency + smallestNodes[1].frequency,
smallestNodes[0], smallestNodes[1]))
```

```
class TreeNode:
def __init__(self, letter, frequency, leftNode=None, rightNode=None):
self.letter = letter
self.frequency = frequency
self.leftNode = leftNode
self.rightNode = rightNode

def __lt__(self, other):
return self.frequency < other.frequency
```

3. Tworzenie słownika kodowego. Również w tym przypadku skorzystaliśmy ze słownika oferowanego w pythonie. Jego zawartość wypełniamy poprzez rekurencyjne przejście drzewa od korzenia w dół. Dla węzłów znajdujących się po lewej stronie dodawane do kodu jest 0 a dla tych po prawo dodawane jest 1.

```
def calculateCodedLetters(self, code, node):
if node.letter is not None:
self.coddedLetters[node.letter] = code
return
self.calculateCodedLetters(code + "0", node.leftNode)
self.calculateCodedLetters(code + "1", node.rightNode)
```

4. Zakodowanie tekstu za pomocą sporządzonego słownika

```
def getCodedText(self):
   encodedText = ""
for letter in self.text:
      encodedText += self.coddedLetters[letter]
return encodedText
```

5. Aby zdekodować zakodowany plik konieczne było odwrócenie słownika. Dekodowanie polega na przejściu przez wszystkie znaki zakodowanego tekstu i jeżeli dana sekwencja znaków występuje w odwróconym słowniku to oznacza, że te znaki zastępujemy znakiem który im odpowiada. W ten sposób postępujemy dopóki nie skończy się zakodowany tekst.

```
self.reversedCoddedLetters = {value: key for (key, value) in self.coddedLetters.items()}
def decodeText(self, codedText):
   code = ""
decodedText = ""
for i in codedText:
      code += i
if code in self.reversedCoddedLetters:
         decodedText += self.reversedCoddedLetters[code]
         code = ""
return decodedText
```

6. Do przesłania plików zostały użyte gniazda sieciowe – kod po stronie serwera wysyła liczbę zer do usunięcia oraz dwa pliki.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 1234))
s.listen(5)

print('Server listening....')

while True:
   conn, addr = s.accept()
print('Got connection from', addr)
   conn.send(bytes(str(h.numberZeros), "utf-8"))
```

```
   file = open('textbin.txt', 'rb')
   l = file.read(8)
while (l):
      conn.send(l)
print('Sent ',repr(l))
      l = file.read(8)
   file.close()

   conn.send(bytes('EOF', 'utf-8'))

   file = open('dict.txt','rb')
   l = file.read(8)
while (l):
      conn.send(l)
print('Sent ',repr(l))
      l = file.read(8)
   file.close()

print('Done sending')
   conn.close()
```

7. Kod po stronie klienta – odbiera wysłane przez serwer pliki i zapisuje w swoich plikach.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1234))
receivedMessage = ""
n = s.recv(1)
zeros = n.decode('utf-8')

while True:
print('receiving data...')
   data = s.recv(8)
print('data=', (data))
if not data:
break
receivedMessage += data.decode("ANSI")
print('Successfully get the file')

open('encodedText.txt', 'wb').write(bytes(receivedMessage[:receivedMessage.find('EOF')], 'ANSI'))
open('dict.txt', 'w').write(receivedMessage[receivedMessage.find('EOF')+3:].replace('\n', ""))

h = Huffman()
h.decompressFile('encodedText.txt', 'decodedText.txt', 'dict.txt', zeros)

s.close()
print('connection closed')
```

Podsumowanie wnioski.

Uważamy, że zaimplementowany przez nas algorytm działa poprawnie, ponieważ rzeczywiście zakodowane pliki ulegają bezstratnej kompresji. Wadą tego algorytmu jest konieczność transmisji oprócz pliku również całego słownika albo drzewa, aby mieć możliwość odkodować zakodowany plik. Zaletą jest to, że sama procedura budowy drzewa Huffmana jest szybka i prosta w implementacji.