

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

DANIEL HENRIQUE FERREIRA GOMES

COMPACTADOR BASEADO NA CODIFICAÇÃO DE HUFFMAN

NATAL

2018

1. Introdução

O presente estudo trata da implementação de um compactador baseado na codificação de Huffman. O único desenvolvedor foi o aluno Daniel Henrique Ferreira Gomes, além disso, o código fonte se encontra e disponível em <https://github.com/Danhfg/Huffman-compression>. A linguagem escolhida foi Python3 e as tecnologias utilizadas foram: github, compilador python3, Ubuntu, e jupyter.

O trabalho proposto pela professora Sílvia Maria Diniz Monteiro Maia consistiu em desenvolver um compactador que se baseia na codificação de Huffman.

Além da compactação, também se fez necessário criar um descompactador.

2. Desenvolvimento

2.1. Implementação da codificação de Huffman

A implementação da codificação de Huffman seguiu a orientação dada nas instruções do projeto, no caso, primeiro cria uma lista de nós individuais e sem filhos, depois vai juntando em um novo nó os dois nós com menor prioridade e os removendo da lista, posteriormente o novo nó adicionando o novo nó e os passos são repetidos até que reste somente a raiz. A Figura 1 ilustra a implementação da codificação de Huffman enquanto a Figura 2 apresenta uma função auxiliar que busca os dois nós com menor prioridade.

```
nodes = []
#Criar lista de árvores com um nó
for chave in tabela_inicial:
    nodes.append(Node(None, None, tabela_inicial[chave], chave))

#Algoritmo de Huffman
while len(nodes) > 1:
    menores = menores_nodes(nodes)
    node_pai = Node(menores[0], menores[1],
                    menores[0].get_apar() + menores[1].get_apar(),
                    menores[0].get_apar() + menores[1].get_apar())
    nodes.remove(menores[0])
    nodes.remove(menores[1])
    nodes.append(node_pai)
```

Figura 1 -Codificação de Huffman

```
def menores_nodes(nodes):
    #m1, m2 = float('inf'), float('inf')
    m1 = Node(None, None, float('inf'), 'inf')
    m2 = Node(None, None, float('inf'), 'inf')
    for x in nodes:
        if x.get_apar() < m1.get_apar():
            m1, m2 = x, m1
        elif x.get_apar() < m2.get_apar():
            m2 = x
    return m1, m2
```

Figura 2 -Encontrar os dois nós com menores prioridades

2.2. Estrutura do código

Apenas duas classes foram implementadas, a classe Node e a classe Tree. Como é possível observar pela Figura 3, a classe node consiste em um nó com dois filhos, um à direita e outro à esquerda, a quantidade de aparições de um caractere e o caractere em si.

```

class Node:
    e = None
    d = None
    apar = 0
    char = ''

    def __init__(self, e, d, apar, char):
        self.e = e
        self.d = d
        self.apar = apar
        self.char = char

    def get_apar(self):
        return self.apar
    def get_char(self):
        return self.char
    def get_e(self):
        return self.e
    def get_d(self):
        return self.d

```

Figura 3 - Classe Node

Enquanto isso, a Figura 4 mostra que a classe Tree consiste apenas em uma árvore com um nó raiz.

```

class Tree:
    node_root = None

    def __init__(self, nr):
        self.node_root = nr
    def get_root(self):
        return self.node_root
    def set_root(self, node):
        self.node_root = node

```

Figura 4 - Classe Tree

2.3 Criação do arquivo compactado

O primeiro passo para a criação do arquivo compactado é a criação da tabela de codificação, no caso, consiste em um dicionário (chave, valor), onde a chave do dicionário é o caractere e o valor é binário(baseado na árvore), como ilustra a Figura 5.

```
def tabela_pre(node, caminho, t):
    if(node != None):
        if(type(node.get_char()) == str):
            t[node.get_char()] = ''.join(caminho)
        else:
            caminho_e = caminho[:]
            caminho_d = caminho[:]
            caminho_e.append('0')
            caminho_d.append('1')
            tabela_pre(node.get_e(), caminho_e, t)
            tabela_pre(node.get_d(), caminho_d, t)
```

Figura 5 - Criação da tabela em pré-ordem

Com a tabela, se faz necessário a criação de um cabeçalho que guiará o descompactador para a criação da árvore de Huffman. A implementação da criação do cabeçalho está representado na Figura 6.

```
def cab(node, cabecalho_final, tabela_final):
    if(node != None):
        if(type(node.get_char()) == str):
            cabecalho_final[0] += '1'
            if (node.get_char() == 'EOF'):
                cabecalho_final[0] += '1111111'
        else:
            cabecalho_final[0] += '{:0>7}'.format(bin(int.from_bytes(node.get_char().encode(), 'big'))[2:])
    else:
        cabecalho_final[0] += '0'
    cab(node.get_e(), cabecalho_final, tabela_final)
    cab(node.get_d(), cabecalho_final, tabela_final)
```

Figura 6 - Criação do cabeçalho

Após isso, o cabeçalho em binário é concatenado com o texto em binário(criado a partir da tabela de codificação) e essa informação é salva conforme mostra a Figura 7.

```

def salvar_arquivo(arquivo):
    with open('compact.bin', 'wb') as f:
        i = 0
        while (i < len(arquivo)):
            if(i + 8 > len(arquivo)):
                n = int('{:0<8}'.format(arquivo[i:]), 2)
                f.write(n.to_bytes((n.bit_length() + 7) // 8, 'big'))
                #print(n.to_bytes((n.bit_length() + 7) // 8, 'big'))
            else:
                n = int(arquivo[i:i+8], 2)
                if (n == 0):
                    f.write(b'\x00')
                    #print(b'\x00')
                else:
                    f.write(n.to_bytes((n.bit_length() + 7) // 8, 'big'))
                    #print(n.to_bytes((n.bit_length() + 7) // 8, 'big'))
            i += 8

```

Figura 7 - Criação do cabeçalho

2.4. Descompactador

O implementação do descompactador é apresentada na Figura 8, nela podemos ver vai haver uma pilha com os nós que são criados, de modo que quando o caractere atual é igual a '0' o nó é um nó não folha, caso contrário é um nó folha.

Caso o algoritmo encontre um nó não folha, a leitura contínua para o próximo caractere e o nó é adicionado na árvore. Quando uma folha é identificada é os próximos 7 caracteres são lidos, vistos que eles representam apenas um caractere em código ascii. Quando não há mais nós na pilha significa que o cabeçalho terminou.

Com o cabeçalho lido, basta ler o resto do arquivo com base na árvore de Huffman criada que os caracteres serão descompactados. A função responsável por isso é a função `decode_char()` ilustrada na Figura 9.

O resultado obtido em `decode_char()` é justamente o texto acrescido do EOF, marcador de fim de arquivo, com isso, basta retirar este marcador e salvar a string em um arquivo, como mostra a Figura 10.


```

def decode(string_cod):
    if(string_cod != ""):

        stack = []

        node = Node(None, None, 'raiz', 0)
        #new_node = (None, None, 'raiz', 'raiz')
        stack.append(node)
        tree = Tree(node)

        index = 1
        while (len(stack) != 0 and index < len(string_cod)):
            if (string_cod[index] == '0'):
                node = Node(None, None, 'n-folha', 0)
            elif(string_cod[index] == '1'):
                aux = string_cod[index + 1: index + 8];
                if(aux == '1111111'):
                    #print(aux, 'EOF')
                    node = Node(None, None, 'folha', 'EOF')
                    index+= 7
                else:
                    convert = int(aux, 2)
                    #print(aux, convert, end=' ')
                    convert = convert.to_bytes(
                        [(convert.bit_length() + 7) // 8, 'big']).decode()
                    #print(convert)
                    node = Node(None, None, 'folha', convert)
                    index+= 7
            if(stack[-1].get_e() == None):
                stack[-1].set_e(node)
            else:
                stack[-1].set_d(node)
                stack.pop()
            if(type(node.get_char()) == int):
                stack.append(node)
            index+=1
        return (tree.get_root(), index)

```

Figura 8 - Método para descobrir se uma ABB é cheia

```

def decode_char(root, pos, string):
    if(root != None):
        if(root.get_e() == None and root.get_d() == None):
            return str(root.get_char())
        pos[0] = pos[0] + 1
        if (string[pos[0]] == '0' ):
            return decode_char(root.get_e(), pos, string)
        else:
            return decode_char(root.get_d(), pos, string)

```

Figura 9 - Método que decodifica o resto do texto

```
def salvar_arquivo_descompactado(arquivo):
    with open('descompact.txt', 'w') as f:
        f.write(arquivo)
```

Figura 10 - Método que decodifica o resto do texto

2.6. Exemplo de execução

Um exemplo de execução do compactador e descompactador é mostrado na Figura 11, nela podemos ver que a tabela de compactação para o arquivo teste_1.txt foi criada corretamente assim como o arquivo binário(Figura 12). Além disso, podemos observar que o descompactador criou um arquivo igual ao arquivo compactado através do comando diff do linux.

```
daniel@daniel-Inspiron-5437:~/Documentos/EDB2/Huffman-coding$ python3 compressio
n.py teste_1.txt
teste_1.txt
Tabela de codificação: {'d': '00000', ',': '000010', 'v': '000011', 'r': '0001',
'e': '001', 'p': '01000', 'b': '010010', 'q': '010011', 'l': '0101', 'n': '011
0', 'f': '0111000', 'h': '0111001', 'g': '0111010', '\n': '011101100', 'EOF': '0
11101101000', 'C': '011101101001', 'j': '01110110101', 'S': '011101101100', 'L':
'0111011011010', 'T': '0111011011011', 'A': '0111011011100', 'R': '011101101110
1', 'P': '0111011011110', 'N': '0111011011111', '.': '01110111', 'm': '01111', '
': '100', 't': '1010', 'u': '1011', 's': '1100', 'a': '1101', 'c': '11100', 'o'
: '11101', 'i': '1111'}
Arquivo compactado criado com sucesso (compact.txt)!!!!
daniel@daniel-Inspiron-5437:~/Documentos/EDB2/Huffman-coding$ python3 decompress
ion.py compact.bin
Arquivo descompactado com sucesso(descompact.txt)!!!!
daniel@daniel-Inspiron-5437:~/Documentos/EDB2/Huffman-coding$ diff teste_1.txt d
escompact.txt
daniel@daniel-Inspiron-5437:~/Documentos/EDB2/Huffman-coding$
```

Figura 11 - execução compactador e descompactador no terminal()

compact.bin	teste_1.txt	compression.py
0723	a3c7	b394
f2ee	3c1c	5ede
6500	7373	
833b	fe29	d4c1
666b	5756	77b8
fb4e	1f57	
a74d	2755	0de7
b1f6	f1b2	17db
4723	072c	
533a	3ebb	0a83
ddcc	be51	5c8f
8cf2	b0a5	
5e9c	fb0	a03c
f47c	6795	964c
78f8	cf2b	
1898	5caf	41cb
7afe	4b6a	635f
da94	e551	
710c	6f45	9bd9
c660	f87a	7a65
cf40	f7dc	
3d31	cde9	61ef
cbc0	b2d0	d12f
d8f0	1bf0	
f7e5	e674	bd44
7de1	d996	b6dd
f163	9bcb	
6b5e	3f93	c12d
a98d	7f6a	4668
b7eb	894e	
5aaa	e43d	ce8b
eff0	1bef	2da9
8d7f	6a40	
6fbc	6058	8793
21ef	cbc6	fdb2
8991	7447	
2552	df0e	c5f6
82a5	81ef	b87a
6310	f7ed	
e5fe	03f2	f1cd
d852	02aa	6d17
2f1b	217d	

Figura 12 - execução compactador e descompactador no terminal

A compactação se mostrou bastante eficiente, uma vez que, como mostra a Figura 13, o arquivo original possuía 7,4 kB enquanto o arquivo compactado possuía 3,9kB.



	descompact.txt	7,4 kB	Texto	23:30
	compact.bin	3,9 kB	Binário	23:29

Figura 13 - Tamanho dos arquivos

3. Conclusão

Podemos concluir que a compactação baseada na codificação de Huffman foi bastante eficiente, visto que como mostrado na Figura 13, o arquivo original ocupa quase o dobro do tamanho do arquivo compactado, aproximadamente 1.8974.