Relatório Trabalho Prático 2

Alexandre Rodrigues nº 2022249408

Rodrigo Borges nº 2022244993

Pedro Pires nº 2022247126

Índice

Introdução

Nas últimas semanas, desenvolvemos um programa para descompactar dados usando o algoritmo LZ77 e códigos de Huffman. Começámos por ler o formato do bloco e armazenar os comprimentos dos códigos. Posteriormente, convertemos esses comprimentos em códigos de Huffman, lemos e armazenamos os comprimentos dos alfabetos, determinamos os códigos de Huffman e implementamos funções para a descompactação. No final, gravámos os dados descompactados num ficheiro, referenciando a estrutura gzipHeader. Este projeto combina teoria de compressão de dados com implementações práticas, num processo ao longo de cinco semanas.

Exercício 2

No exercício 2 implementamos o código de forma a criar uma lista chamada ‘comp’ e a preenche-la com comprimentos de códigos gerados pela função ‘gz.comps’ com o argumento HCLEN , ‘gz.comps(HCLEN)’ , ela retorna um array contendo valores de 3 em 3 bits, representando os códigos Huffman gerados.

```
comp = []
comp = gz.comps(HCLEN)
```

Exercício 3

Na função 'count', inicializamos um array 'count' com zeros e tamanho 'n', convertendo-o para o tipo 'uint16'. Utilizando dois ciclos, percorremos 'count' e 'comp', contabilizando as ocorrências de cada valor 'i' em 'comp' e incrementando o respetivo contador em 'count[i]'. Após a contagem, exibimos o array 'count' e o retornamos. Esta função é essencial para determinar a frequência de cada elemento em 'comp', sendo utilizada na função 'huffman' para gerar os códigos Huffman.

Na função 'huffman', após inicializar variáveis e criar um array 'huffman_array', usamos os resultados da função 'count' para gerar códigos Huffman. Percorremos 'count', acumulando valores em 'cod' e utilizando-os para calcular os códigos Huffman. Depois, exibimos 'huffman_array', criamos uma instância 'hft' da classe HuffmanTree e usamo-la para construir a árvore Huffman. Iteramos sobre 'comp', convertendo os códigos para binário e adicionando nós à árvore. A árvore resultante é retornada. Esta função integra a lógica de codificação Huffman, utilizando a contagem de ocorrências obtida pela função 'count'.

```
def count(self,comp,n):
    count = np.zeros(n)
    count = count.astype("uint16")
    for i in range(len(count)):
        for j in range(len(comp)):
            if comp[j] == i:
                count[i]+=1
    print(count)
    return count
```

```
def huffman(self,comp,count,n):
    cod = 0
    count[0] = 0
    huffman_array = np.zeros(n)
    huffman_array = huffman_array.astype("uint16")
    for i in range(1,len(count)):
        cod += count[i-1]
        if(i>2):
            cod = np.left_shift(cod,1)
            huffman_array[i] = cod
    print(huffman_array)
    hft = HuffmanTree()
    i = 0
    for j in comp:
        if(j!=0):
            huffmancode = (format(huffman_array[j],'b')).zfill(j)
            hft.addNode(huffmancode,i,True)
            huffman_array[j]+=1
        i+=1
    return hft
```

Exercício 4

Inicialmente, chamamos a função 'gz.search' com os argumentos 'HLIT + 257' e 'hft', atribuindo o seu resultado à variável 'lit'. Esta função tem como finalidade a geração de um array de comprimentos, recorrendo à árvore de Huffman. Durante este procedimento, ocorre a leitura de bits, a navegação cuidadosa pela árvore para reconstrução dos dados, a manipulação de códigos especiais e o preenchimento do array em questão.

```
lit = gz.search(HLIT + 257,hft)
print(lit)
```

Exercício 7

A função 'calcula_bits' recebe três parâmetros: 'start_pos', 'i' e 'pos'. Inicializa variáveis como 'cur_pos', 'bits', 'count' e 'length' e entra num loop while, que persiste até que 'cur_pos' atinja 'pos'. A cada iteração, a função incrementa 'cur_pos', adiciona (2 ** bits) ao comprimento 'length' e incrementa 'count'. Condições específicas são verificadas para determinar se é necessário atualizar o número de bits. Se a condição for atendida, o número de bits é incrementado, e 'count' é reiniciado. Ao atingir a posição desejada (cur_pos == pos), a função verifica se há bits restantes (bits > 0). Se positivo, a função chama 'self.readBits(bits)' para ler os bits remanescentes e os adiciona ao comprimento 'length'.

A função descomprimir recebe duas árvores de Huffman, ‘hft_HLIT’ e ‘hft_DIST’. No início, o código inicializa variáveis essenciais e entra num loop principal que coordena o processo de descompressão. Dentro desse loop, um segundo loop, aninhado, é dedicado ao tratamento de valores literais/comprimento. Cada bit é lido individualmente, e sua posição na árvore ‘hft_HLIT’ é identificada. Consoante o valor da posição, diferentes ações são desencadeadas. Se a posição for menor que 256, o valor é interpretado como um literal e adicionado ao array resultante. Caso a posição seja igual a 256, a descompressão é considerada concluída para o bloco atual. Se a posição for maior que 256, o código calcula o comprimento por meio de uma função específica e entra em outro loop para tratar os valores de distância.

Dentro do loop de distância, o código lê bits, encontra a posição correspondente na árvore 'hft_DIST', reinicia o nó e calcula a distância. Em seguida, os valores do array de histórico são copiados com base na distância calculada. Após concluir a descompressão para um bloco de comprimento/distância, as variáveis são reiniciadas para a próxima iteração. Este processo se repete até que a descompressão seja concluída para todos os dados fornecidos.

```
def calcula_bits(self,start_pos,i,pos):
    cur_pos = start_pos
    bits=0
    count = 0
    length = 0
    while(cur_pos!=pos):
        cur_pos+=1
        length += (2**bits)
        count+=1
        if(bits == 0 and count >=i*2)or(bits!=0 and count >=i):
            bits+=1
            count = 0
    if(bits>0):
        length += self.readBits(bits)
    return length
```

```
def descomprimir(self,hft_HLIT, hft_DIST):
    array = np.array([])
    array = array.astype("uint16")
    terminate_LIT = False
    terminate_DIST = False
    notFinished = True
    i=0
    length=0
    while (notFinished):
        while not terminate_LIT:
            nextBit = str(self.readBits(1))
            pos = hft_HLIT.nextNode(nextBit)
            if pos != -2:
                terminate_LIT = True
                if (pos<256):
                    array = np.append(array,[pos],axis = 0)
                    i+= 1
                if (pos==256):
                    notFinished=False
                if (pos>256):
                    length = 3 + self.calcula_bits(257,4,pos)
                    while not terminate_DIST:
                        nextBit = str(self.readBits(1))
                        next_pos = hft_DIST.nextNode(nextBit)
                        if next_pos != -2:
                            terminate_DIST = True
                    hft_DIST.resetCurNode()
                    terminate_DIST = False
                    distance = 1 + self.calcula_bits(0,2,next_pos)
                    for i in range(length):
                        array = np.append(array,[array[-distance]],axis=0)
        terminate_LIT = False
        hft_HLIT.resetCurNode()
    return array
```

Conclusão

Com grande esforço ao longo destas cinco semanas, conseguimos desenvolver um programa sólido para descomprimir dados usando o algoritmo LZ77 e códigos de Huffman. Iniciámos com a leitura do formato do bloco e o armazenamento dos comprimentos dos códigos, seguido pela conversão destes comprimentos em códigos de Huffman. As etapas seguintes envolveram a leitura e armazenamento dos comprimentos dos alfabetos, a determinação dos códigos de Huffman e a implementação das funções necessárias para a descompressão.

No final deste desafio, lográmos gravar os dados descomprimidos num ficheiro, respeitando a estrutura da gzipHeader. Este percurso exigiu um esforço notável, proporcionando uma compreensão aprofundada da teoria de compressão de dados e a sua aplicação prática. A combinação destas diferentes etapas ao longo das semanas culminou numa solução eficaz e funcional para a descompressão de dados comprimidos.