UNIVERSIDADE D
COIMBRA

Licenciatura em Engenharia Informática

Teoria da Informação

Trabalho Prático nº2

Descompactação de Ficheiros “gzip”

Discentes:

Beatriz Costa Pereira Monteiro – 2021234863

Filipe Freire Soares – 2020238986

Luís Gonçalo Teixeira Pereira – 2021230244

Coimbra, 2022/2023

Índice

Índice de Figuras

Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular Teoria da Informação, em que nos foi proposta a realização de um trabalho prático com o objetivo de implementar um descodificador do algoritmo deflate, descompactando blocos comprimidos com códigos de Huffman dinâmicos.

A realização deste trabalho permitiu-nos responder a questões fundamentais relacionadas com a codificação usando árvores de Huffman e dicionários LZ77.

Este relatório serve para, de uma forma pormenorizada, não só explicar o código, mas também compreender a matéria dada.

Gzip

O ficheiro gzip é um formato de arquivo que contém blocos de dados compactados. O formato de cada bloco é especificado no seguimento do relatório. Estes aparecerem uns a seguir aos outros, de acordo com a sua ordem no ficheiro, sem qualquer informação adicional entre cada membro.

Este tipo de formato é usado, por exemplo, na internet para economizar a largura de banda. O interessante acerca do gzip é que a descompactação do mesmo pode ser implementada como um algoritmo de streaming, o que o torna um bom candidato para compactação em protocolos da web.

Exercício 1

O exercício 1 consiste no desenvolvimento de uma função que leia o formato do bloco, ou seja, que devolva o valor correspondente a HLIT, HDIST e HCLEN, de acordo com a estrutura de cada bloco.

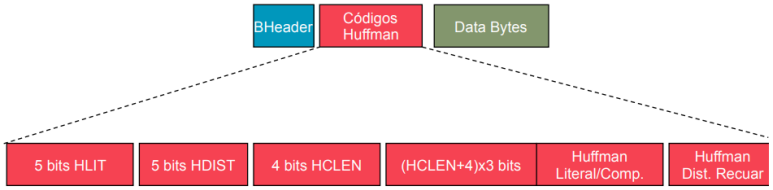

Figura 1- Estrutura de cada bloco.

Para a realização deste exercício, foi necessário implementar a função "readBlockFormat".

```
def readBlockFormat(self):
    #HLIT -> 257 - 286
    HLITValue = self.readBits(5)
    HLITValue = HLITValue + 257

    #HDIST -> 1 - 32
    HDISTValue = self.readBits(5)
    HDISTValue = HDISTValue + 1

    #HCLEN -> 4 - 19
    HCLENValue = self.readBits(4)
    HCLENValue = HCLENValue

    return HLITValue, HDISTValue, HCLENValue
```

Figura 2- Função "readBlockFormat".

A função lê cada bloco separadamente, segundo o número de bits que cada um possui. Desta forma, para cada bloco, são lidos os n bits da seguinte forma:

HLIT = número de códigos de comprimento literal – 257 (257 – 286);

HDIST = número de códigos de distância – 1 (1 – 32) (os códigos de distância 30 – 31 não aparecem em blocos compactados, mas participam da construção da árvore Huffman dos códigos de distância);

HCLEN = número de comprimentos de código – 4 (4 – 19).

Exercício 2

No exercício 2, tem-se como objetivo criar um método que insira num array, os comprimentos dos códigos de cada símbolo presente no alfabeto de comprimentos de códigos com base em HCLEN.

É importante referir que as sequências de 3 bits a ler correspondem à ordem 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15 no array. Por isso, foi criado um array com essa sequência para que seja possível inserir no array os comprimentos dos códigos de cada símbolo na ordem correta.

Deste modo, foi criado o método “codeLengthsValue” que recebe como parâmetros o bloco HCLEN e o array com a sequência de 3 bits a ler anteriormente descrito. Esta função retorna o array pedido no enunciado.

```
def codeLengthsValue(self, HCLEN, codeLengthsOrder):
    codeLengthsArray = [0] * 19
    HCLENValue = 0

    for i in range(HCLEN + 4):
        HCLENValue = self.readBits(3)
        codeLengthsArray[codeLengthsOrder[i]] = HCLENValue

    return codeLengthsArray
```

Figura 3- Função "codeLengthsValue".

Neste método foi percorrido o bloco HCLEN + 4 e guardado numa variável os 3 bits lidos.

Para respeitar a sequência de 3 bits a ler, foi colocado o valor da variável na posição correspondente ao valor que se encontra no array da sequência, passado como parâmetro, no respetivo índice.

Exercício 3

No ponto 3 é proposta a criação de um método que permita a conversão dos comprimentos dos códigos, determinados na alínea anterior com base em HCLEN, em códigos de Huffman. Este mesmo método será posteriormente usado na realização do ponto 6.

Para tal, desenvolvemos o método codeLengthsHuffman() que recebe como parâmetro um array contendo os comprimentos dos códigos nas respetivas posições.

```
def codeLengthsHuffman(self, codeLengthsArray):
    huffmanCodesDic = {}

    blCountDic = {}
    for i in codeLengthsArray:
        if (i not in blCountDic.keys()):
            blCountDic.setdefault(i, 1)

        else:
            blCountDic[i] += 1

    blCountDic = dict(sorted(blCountDic.items()))

    blCountKeys = []
    for i in blCountDic.keys():
        blCountKeys.append(i)

    maxKeyValue = 0
    maxKeyValue = blCountKeys[len(blCountKeys) - 1]

    blCountAux = [0] * (maxKeyValue + 1)
    for i in blCountDic.keys():
        blCountAux[i] = blCountDic[i]

    nextCode = [0] * (maxKeyValue + 1)

    code = 0
    blCountAux[0] = 0
    for i in range(1, len(nextCode)):
        code = (code + blCountAux[i - 1]) << 1
        nextCode[i] = code

    huffmanCodesInt = []
    for i in range(len(codeLengthsArray)):
        lenCode = codeLengthsArray[i]

        if (lenCode != 0):
            huffmanCodesInt.append(nextCode[lenCode])
            nextCode[lenCode] += 1

    codeLengthsAux = [i for i in codeLengthsArray if i != 0]

    huffmanCodes = []
    for i in huffmanCodesInt:
        huffmanCodes.append(format(i, "#010b"))

    for i in range(len(huffmanCodes)):
        huffmanCodes[i] = huffmanCodes[i][2 : len(huffmanCodes[i])]

    for i in range(len(codeLengthsAux)):
        if (codeLengthsAux[i] < 8):
            zeroPosition = 8 - codeLengthsAux[i]
            huffmanCodes[i] = huffmanCodes[i][zeroPosition : len(huffmanCodes[i])]

    codeSymbols = [i for i in range(len(codeLengthsArray)) if codeLengthsArray[i] != 0]
    for i in range(len(codeSymbols)):
        huffmanCodesDic.setdefault(codeSymbols[i], huffmanCodes[i])

    return huffmanCodesDic
```

Figura 4- Função "codeLengthsHuffman"

Primeiramente é inicializado um dicionário blCountDic, o array é percorrido e os seus valores são adicionados às keys onde o seu respetivo value é inicializado a 1. Sempre que um valor do array já se encontre registado nas keys do dicionário, o seu value é incrementado. Após percorrer todo o array, o dicionário é ordenado por ordem crescente de key, ou seja, por ordem crescente de comprimento de código, obtendo-se assim um dicionário com o número de ocorrências de todos os comprimentos dos códigos.

De seguida, são guardados num array os comprimentos dos códigos ordenados por ordem crescente de forma a obtermos o valor do maior comprimento de código. Esse valor é usado de forma que seja possível inicializar os arrays blCountAux e nextCode com um tamanho definido de 0's. O array blCountAux irá conter todos os values do dicionário, isto é, os números de ocorrências de cada comprimento de código. Logo em seguida, o array nextCode irá conter todos os menores valores dos códigos de Huffman em inteiros para cada comprimento de código.

De forma a obtermos todos os códigos de Huffman, inicialmente em inteiros, selecionamos todos os comprimentos diferentes de zero, adicionamos o respetivo valor no array nextCode valor ao array huffmanCodesInt e incrementamos o valor no array nextCode.

Por fim, de modo a convertermos todos os códigos de Huffman de inteiros para sequências de 0's e 1's, começamos por adicionar todos os valores, convertidos para binário e com pelo menos 8 bits de comprimento, ao array huffmanCodes e logo depois são removidos todos os 0's em excesso de acordo com os comprimentos dos códigos. Todos estes códigos de Huffman são adicionados a um dicionário com os respetivos símbolos, sendo este retornado pelo método.

Após a conversão de todos os comprimentos de códigos em códigos de Huffman, é criada a respetiva árvore de Huffman com recurso ao método generateTree().

Exercício 4

O quarto exercício pede para criar um método que leia e armazene num array os HLIT + 257 comprimentos dos códigos referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman elaborado no exercício anterior.

Para a resolução deste exercício, foi necessário criar a função "literalLengthValues" que recebe como parâmetros a árvore de Huffman com os códigos de Huffman do alfabeto de comprimentos de códigos, realizada no exercício anterior, e o bloco que é pedido, ou seja, o bloco HLIT. Esta função retorna um array que armazena os comprimentos dos códigos pedidos no enunciado.

```
def literalLengthValues(self, hft, HType):
    count = 0
    array = [0] * HType

    while count < HType:
        bit = self.readBits(1)
        pos = hft.nextNode(str(bit))

        if(pos >= 0):
            if(pos == 16):
                repeat = 3
                bit = 0

                for i in range(2):
                    bits = self.readBits(1)
                    bits = bits << i
                    bit = bit | bits

                repeat += bit

                for i in range(repeat):
                    array[count] = array[count-1]
                    count += 1

            elif(pos == 17):
                repeat = 3
                bit = 0

                for i in range(3):
                    bits = self.readBits(1)
                    bits = bits << i
                    bit = bit | bits

                repeat += bit

                for i in range(repeat):
                    array[count] = 0
                    count += 1

            elif(pos == 18):
                repeat = 11
                bit = 0

                for i in range(7):
                    bits = self.readBits(1)
                    bits = bits << i
                    bit = bit | bits

                repeat += bit

                for i in range(repeat):
                    array[count] = 0
                    count += 1

            else:
                array[count] = pos
                count += 1

            hft.resetCurNode()

    return array
```

Figura 5- Função "literalLengthValues".

É importante referir que alguns códigos requerem a leitura de alguns bits extra (nomeadamente os índices 16, 17 e 18 do alfabeto). Para ler e armazenar corretamente os comprimentos de código, foi necessário utilizar o seguinte alfabeto:

Para índices entre 0 e 15, representa-se os respetivos comprimentos de código de 0 a 15.

De seguida, para o índice 16 do alfabeto, é repetido o comprimento de código anterior entre 3 e 6 vezes. Os próximos dois bits determinam quantas vezes se repete o código anterior.

Em relação ao índice 17 do alfabeto, é repetido um comprimento de código de 0 entre 3 e 10 vezes. Os próximos três bits determinam quantas vezes se repete o 0.

Por fim, no que toca ao índice 18 do alfabeto, é repetido um comprimento de código de 0 entre 11 e 138 vezes. Os próximos sete bits determinam quantas vezes se repete o 0.

Exercício 5

No exercício 5, à semelhança do exercício 4, é pedido para criar um método que leia e armazene num array os HDIST + 1 comprimentos de códigos referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman, elaborado no exercício 3.

Para a resolução deste exercício, foi usada a função “literalLengthValues”, criada no exercício anterior, que recebe como parâmetros a árvore de Huffman com os códigos de Huffman do alfabeto de comprimentos de códigos, mas, desta vez, o bloco pedido é o HDIST, por isso chamamos a mesma função, mas alteramos o bloco. Mais uma vez, esta função retorna um array que armazena os comprimentos dos códigos pedidos no enunciado.

Exercício 6

No exercício 6, pretende-se determinar e armazenar num array os códigos de Huffman referentes aos alfabetos dos blocos HLIST e HDIST usando o método criado no exercício 3.

Assim sendo, para resolver este exercício, foi necessário recorrer à função efetuada no exercício 3, ou seja, o método "codeLenghtsHuffman".

Além disso, foi necessário criar a função "convertToArray", responsável por retornar um array com os códigos de Huffman.

```
def convertToArray(self, huffmanCodesHType):
    huffmanCodesArray = []

    for i in huffmanCodesHType.values():
        huffmanCodesArray.append(i)

    return huffmanCodesArray
```

Figura 6- Função "convertToArray".

Foi também necessário criar a função "generateTree" que gera a arvore de Huffman e adiciona os códigos de Huffman a cada nó da árvore.

```
def generateTree(self, huffmanCodes):
    hft = HuffmanTree()
    verbose = True

    for i in huffmanCodes.keys():
        hft.addNode(huffmanCodes[i], i, verbose)

    return hft
```

Figura 7- Função "generateTree".

Deste modo, para cada bloco, é usado o método "codeLenghtsHuffman" que é responsável por converter os comprimentos dos códigos em códigos de Huffman.

Posteriormente, é usada a função "convertToArray" para imprimir esses códigos num array, tal como pedido no enunciado.

Por fim, tendo já acesso aos códigos de Huffman, só falta gerar a árvore usando o método anteriormente descrito.

Exercício 7

No exercício 7 é pedido para descompactar os dados comprimidos, com base nos códigos de Huffman e no algoritmo LZ77.

Desta forma, foi necessário criar a função “decompressData” que recebe como parâmetro as árvores de Huffman correspondentes aos blocos HLIT e HDIST. Esta função vai retornar um array com os dados descompactados em código Ascii.

```
def decompressData(self, hftHLIT, hftHDIST):
    outputAscii = []
    pos = 0

    while (pos != 256):
        bit = self.readBits(1)
        pos = hftHLIT.nextNode(str(bit))

        if (pos >= 0):
            if (pos < 256):
                outputAscii.append(pos)

            else:
                size = [0, 0]

                if (255 < pos < 265):
                    size[0] = pos - 254

                elif (pos == 285):
                    size[0] = 258

                else:
                    bitsToRead = ((pos - 265) // 4) + 1
                    aux = 11

                    for i in range(1, bitsToRead):
                        aux += (4 * (2 ** i))

                    aux += self.readBits(bitsToRead)

                    size[0] = aux

                distCode = -2
                while (distCode < 0):
                    distCode = hftHDIST.nextNode(str(self.readBits(1)))

                hftHDIST.resetCurNode()
                if (distCode < 4):
                    size[1] = distCode + 1

                else:
                    bitsToRead= ((distCode - 4) // 2) + 1
                    aux = 5

                    for i in range (1, bitsToRead):
                        aux += (2 * (2 ** i))

                    aux += ((distCode - 4) % 2) * (2 ** bitsToRead) + self.readBits(bitsToRead)

                    size[1] = aux

                start = len(outputAscii) - size[1]
                for i in range(size[0]):
                    outputAscii += [outputAscii[start + i]]

            hftHLIT.resetCurNode()

    return outputAscii
```

Figura 8- Função "decompressData".

Após a leitura de todos os HLIT + 257 e HDIST + 1 comprimentos de códigos referentes a literais/comprimentos, as conversões em códigos de Huffman e a criação das respetivas árvores, foi desenvolvido o método decompressData() responsável pela descompactação dos dados comprimidos. Este método recebe como parâmetros as duas árvores de Huffman correspondentes a HLIT e a HDIST e recorre ao método nextNode() para efetuar a pesquisa de código de forma sequencial, isto é, bit a bit.

Em primeiro lugar inicializamos o array outputAscii que irá ser retornado contendo todos os códigos ASCII dos respetivos caracteres do ficheiro original.

De seguida, a árvore correspondente a HLIT é percorrida bit a bit, através do método nextNode(), tal como mencionado acima, e caso o valor da variável pos seja diferente de 256, indica que o bloco ainda não foi analisado na totalidade. Caso o valor de pos seja superior ou igual a 0 e inferior a 256, o seu valor é adicionado ao array outputAscii, pois corresponde ao código de literais. Caso contrário, se o valor de pos estiver entre 255 e 265, o valor do comprimento irá derivar entre 3 a 10 (inclusive), pelo que subtraímos 254 ao seu valor. Caso o valor de pos seja 285, o seu comprimento será 258 e, caso contrário, se o valor se encontrar entre 264 e 285, será necessário proceder à leitura de bits extra uma vez que o valor do comprimento poderá variar para esses valores da variável pos.

Por fim, será percorrida a árvore correspondente a HDIST, lidos todos os valores de comprimentos de forma idêntica, isto é, bit a bit e serão adicionados os valores ao array outputAscii, que será retornado pelo método.

Exercício 8

Neste exercício, é pedido que os dados descompactados sejam guardados num ficheiro com o nome original.

Para isso, foi criada a função “writeFile” que recebe como parâmetro o array originado no exercício anterior.

É importante referir que cada índice deste array é composto por código Ascii, que posteriormente será convertido no seu respetivo caracter.

```
def writeFile(self, decompressedData):
    fileName = self.gzh.fName

    f = open(fileName, "w")

    dataString = ""
    for i in decompressedData:
        dataString = dataString + chr(i)

    f.write(dataString)

    f.close()

    return
```

Figura 9- Função "writeFile".

Inicialmente foi criada uma variável com o nome do ficheiro previamente guardado para que o novo ficheiro contenha o mesmo nome.

É inicializa a string dataString que irá conter todo o conteúdo a escrever no ficheiro.

De seguida, foi percorrido o array, recebido como parâmetro, e adicionado a essa string os caracteres correspondentes aos códigos de Ascii de cada índice.

Por último, é escrito o conteúdo da string dataString no ficheiro.

Conclusão

Ao longo da realização deste trabalho prático foi possível consolidar diversos conceitos abordados nas aulas Teóricas e Teórico/Práticas, maioritariamente relacionados com o deflate, bem como adquirir sensibilidade para as questões fundamentais relacionadas com codificação usando árvores de Huffman e dicionários LZ77.

É importante referir que, para o desenvolvimento deste trabalho prático foram usados os ficheiros gzip.py e huffmantree.py disponibilizados, apesar de serem aceites outras implementações.

Além disso, foram também usados como referência os documentos de apoio ao trabalho prático de forma a permitir uma correta elaboração do mesmo.

Por fim, podemos concluir que foi colocado em prática, simultaneamente, o trabalho em equipa, a capacidade de analisar e discutir resultados e de otimizar diversos algoritmos.