UNIVERSIDADE D
COIMBRA

Faculdade de Ciências e Tecnologias

TEORIA DA INFORMAÇÃO - LICENCIATURA EM ENGENHARIA INFORMÁTICA

Descompactação de Ficheiros GZIP

Trabalho Realizado por:
Raphael Kuwae e Teodoro Martins

Dezembro, 2024

1 Introdução e objetivos

O propósito deste trabalho é desenvolver um programa em Python capaz de realizar a descompressão de arquivos no formato GZIP. Especificamente, o foco está na implementação de um decodificador que consiga lidar com blocos compactados utilizando códigos de Huffman dinâmicos, conforme o algoritmo *deflate*. Para isso, cada bloco lido exige a definição de três árvores de Huffman, cujos códigos servirão como base para a descompressão. A extração dos dados descompactados utiliza essas árvores em conjunto com o algoritmo LZ77.

2 Funcionamento

2.1 Estrutura dos blocos

Antes de começar, é necessária a compreensão da estrutura de cada bloco de dados (Imagem 2.1.1), esta é constituída por 3 partes:

- Header: apenas 3 bits, o primeiro indica se o bloco atual é o último e os outros 2 indicam o tipo de compressão usado, tendo em conta que este projeto tem como objetivo a descompressão de blocos exclusivamente comprimidos com árvores de Huffman dinâmicas, estes 2 bits não serão relevantes.
- Códigos Huffman: responsável pela definição dos códigos de Huffman que serão usados para a descompactação.
    - HLIT: Determina o número de códigos no alfabeto de literais/comprimentos, com valores entre 257 e 286.
    - HDIST: Define a quantidade de códigos no alfabeto das distâncias, com valores entre 1 e 32.
    - HCLEN: Indica o número de códigos no alfabeto dos comprimentos dos códigos.
    - Após este conjunto de 14 bits, vêm mais (HCLEN+4)*3 bits, cada um destes subconjuntos de 3 bits define o comprimento do código na árvore dos comprimentos de código do seu elemento correspondente na tabela [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15].

    - De seguida, há HLIT+257 códigos de elementos da árvore de comprimentos de código que correspondem aos comprimentos de código dos elementos da árvore dos literais/comprimentos.
    - Finalmente, há HDIST+1 códigos de elementos da árvore de comprimentos de código que correspondem aos comprimentos de código dos elementos da árvore das distâncias.
- Data Bytes: os dados comprimidos que serão descompactados com base nas árvores de Huffman definidas pela secção anterior

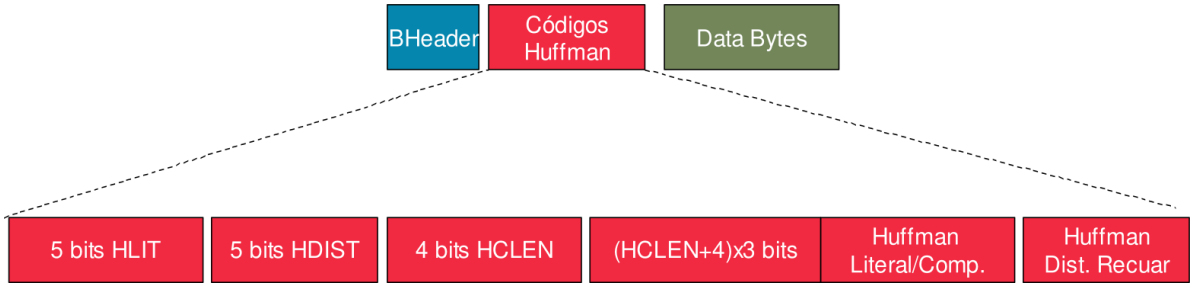

Imagem 2.1.1: Estrutura de um bloco

2.2 Definição dos códigos de Huffman

2.2.1 Leitura de HLIT, HDIST e HCLEN

A primeira tarefa, é, evidentemente, a leitura dos parâmetros HLIT, HDIST e HCLEN, esta é feita através do método `readDynamicBlock` que, por sua vez usa o método `readBits` para ler os 5 bits do HLIT, os 5 do HDIST e os 4 do HCLEN, como é apresentado no código seguinte:

```
def readDynamicBlock (self):
    '''Interprets Dynamic Huffman compressed blocks'''

    HLIT = self.readBits(5)
    HDIST = self.readBits(5)
    HCLEN = self.readBits(4)

    return HLIT, HDIST, HCLEN
```

2.2.2 Leitura dos comprimentos de código para o alfabeto de comprimentos de código

De seguida é feita a leitura de (HCLEN+4)*3 bits, sendo cada subconjunto de 3 bits define o comprimento do código na árvore de comprimentos de código correspondente à posição em que é lido, de acordo com os índices da tabela: [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15]. Isto é feito através do método `storeCLENLengths`.

```
def storeCLENLengths(self, HCLEN):
    '''Stores the code lengths for the code lengths alphabet
    in an array'''

    # Order of lengths in which the bits are read
    idxCLENcodeLens = [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2,
14, 1, 15]
    CLENcodeLens = [0 for i in range(19)]

    # CLENcodeLens[idx] = N translates to: "the code for idx in the code
    # lengths alphabet has a length of N"
    # if N == 0, that indexes' code length is not used
    for i in range(0, HCLEN+4):
        temp = self.readBits(3)
        CLENcodeLens[idxCLENcodeLens[i]] = temp
    return CLENcodeLens
```

2.2.3 Criação de árvores de Huffman baseadas nos comprimentos de código

Tendo em conta que as secções seguintes do bloco vão depender da árvore de Huffman dos comprimentos do código, esta deve ser criada nesta fase do programa. Para tal foi usado o método `createHuffmanFromLens`. O algoritmo utilizado é baseado em duas regras

- Todos os códigos com o mesmo comprimento vão ter valores lexicograficamente consecutivos, por exemplo se ‘D’ e ‘C’ ambos tiverem comprimento 3 e forem os únicos no alfabeto com esse comprimento, o código de ‘C’ seria ‘110’ e o código de ‘D’ seria ‘111’;
- Os códigos mais curtos lexicograficamente vêm sempre antes dos mais longos;

Para construir o algoritmo baseado nestas regras, podemos dividi-lo em duas partes. Primeiro vamos guardar no índice ‘i’ do array `bl_count` o número de códigos de comprimento ‘i’.

```
bl_count = [0 for i in range(max_len+1)]
# Get array with number of codes with length N (bl_count)
for N in range(1, max_len+1):
    bl_count[N] += lenArray.count(N)
```

De seguida, vamos descobrir o 'ponto de partida' para cada comprimento de código, isto é, o valor numérico do código de menor comprimento para cada comprimento, guardando o código mais pequeno para o comprimento 'N', no índice 'N' do array `next_code`, de acordo com o seguinte algoritmo.

```
# Get first code of each code length
code = 0
next_code = [0 for i in range(max_len+1)]
for bits in range(1, max_len+1):
    code = (code + bl_count[bits-1]) << 1
    next_code[bits] = code
```

Finalmente, vamos dar valores numéricos a todos os códigos, incrementando para cada código de um dado comprimento, começando nos códigos definidos no passo anterior, adicionando cada código obtido à árvore de Huffman `htr` e associando esse código ao símbolo correspondente.

```
# Define codes for each symbol in lexicographical order
for n in range(max_symbol):
    # Length associated with symbol n
    length = lenArray[n]
    if(length != 0):
        code = bin(next_code[length])[2:]
        # In case there are 0s at the start of the code,
        # we have to add them manually
        # length-len(code) 0s have to be added
        extension = "0"*(length-len(code))
        htr.addNode(extension + code, n, verbose)
        next_code[length] += 1
```

Por fim, o método devolve a árvore `htr`, e o valor de retorno é guardado na variável `HuffmanTreeCLENs`.

```
# Based on the CLEN tree's code lens, define an huffman tree for CLEN
HuffmanTreeCLENs = self.createHuffmanFromLens(CLENcodeLens, verbose=False)
```

2.2.4 Criação de árvores de Huffman para os alfabetos dosliterais/comprimentos e das distâncias

As árvores de Huffman do alfabeto dos literais/comprimentos e das distâncias, vão ambas ser criadas de uma maneira semelhante à árvore `HuffmanTreeCLENs`, no entanto, ao contrário desta, na qual os comprimentos de código foram lidos diretamente em conjuntos de 3 bits, os comprimentos dos códigos das árvores dos literais/comprimentos e a das distâncias vão ser apresentados em forma de códigos da árvore `HuffmanTreeCLEN`.

O método responsável por esta leitura é o `storeTreeCodeLens`, este recebe 2 parâmetros, o tamanho da árvore cujos comprimentos de código serão guardados e também a árvore dos comprimentos de código (`HuffmanTreeCLEN`).

Como ambas as secções do bloco que contém os comprimentos de código da árvore dos literais/comprimentos e a da árvore das distâncias não têm um tamanho 1xo, como a anterior, que tinha sempre tamanho (HCLEN+4)*3, a leitura terá de ser feita até os arrays de comprimentos de código (`treeCodeLens`) terem o tamanho adequado, ou seja, HLIT + 257 e HDIST + 1, respetivamente. Isto quer dizer que o código do método terá de estar contido no seguinte ciclo while:

```
while (len(treeCodeLens) < size):
```

Cada ciclo começa por colocar o nó atual na raíz de `HuffmanTreeCLEN` e ler bits sequencialmente até encontrar uma folha da árvore através do método `nextNode` da classe `HuffmanTree`, esta verificação é feita com base na ideia de que `nextNode` devolve -1 quando o código atual é inválido e -2 quando o código atual não é uma folha.

```
# Sets the current node to the root of the tree
CLENTree.resetCurNode()
found = False
# While reading, if a leaf hasn't been found, keep searching bit by bit
while(not found):
    curBit = self.readBits(1)
    code = CLENTree.nextNode(str(curBit))
    if(code != -1 and code != -2):
        found = True
```

Quando uma folha é encontrada, não basta colocar o símbolo correspondente à folha no array de comprimentos de código pois existem alguns caracteres especiais com regras específicas que se devem ter em conta, nomeadamente:

- 16: Lê mais 2 bits que definem quantas vezes será copiado o caracter lido no ciclo anterior, o número de cópias pode variar entre 3 e 6.

- 17: Lê mais 3 bits que definem quantas vezes será inserido ‘0’, o número de inserções pode variar entre 3 e 10.
- 18: Lê mais 7 bits que definem quantas vezes será inserido ‘0’, o número de inserções pode variar entre 11 e 138.

Estes caracteres especiais são verificados e tratados de acordo as regras estipuladas pelo algoritmo:

```
if(code == 18):
   amount = self.readBits(7)
   # According to the 7 bits just read, set the following 11-138 values on
the length array to 0
   treeCodeLens += [0]*(11 + amount)
if(code == 17):
   amount = self.readBits(3)
   # According to the 3 bits just read, set the following 3-10 values on the
length array to 0
   treeCodeLens += [0]*(3 + amount)
if(code == 16):
   amount = self.readBits(2)
   # According to the 2 bits just read, set the following 3-6 values on the
length array to the latest length read
   treeCodeLens += [prevCode]*(3 + amount)
elif(code >= 0 and code <= 15):
   # If a special character isn't found, just set the next code length to the
value found
   treeCodeLens += [code]
   prevCode = code
```

Finalmente, o método devolve `treeCodeLens` e o valor de retorno é chamado no respetivo array de comprimentos de código.

```
LITLENcodeLens = self.storeTreeCodeLens(HLIT + 257, HuffmanTreeCLENs)
DISTcodeLens = self.storeTreeCodeLens(HDIST + 1, HuffmanTreeCLENs)
```

As árvores de Huffman de literais/comprimento e distâncias serão, posteriormente, criadas de maneira análoga a `HuffmanTreeCLENs`.

```
HuffmanTreeLITLEN = self.createHuffmanFromLens(LITLENcodeLens, verbose=False)
HuffmanTreeDIST = self.createHuffmanFromLens(DISTcodeLens, verbose=False)
```

2.3 Leitura dos dados

Após estarem definidas `HuffmanTreeLITLEN` e `HuffmanTreeDIST`, resta apenas ler a parte final do bloco que contém os dados comprimidos. Evidentemente, esta descompressão será feita de acordo com o algoritmo LZ77 no método `decompressLZ77`.

O código do caracter 256 na árvore `HuffmanTreeLITLEN` é um caracter especial que indica o final do bloco, tendo isto em conta, a leitura desta parte do bloco será contida no seguinte ciclo while:

```
        while(codeLITLEN != 256):
```

Cada ciclo começa por colocar o nó atual na raíz de `HuffmanTreeLITLEN`, e ler bits sequencialmente até encontrar uma folha dessa árvore.

```
# Resets the current node to the tree's root
HuffmanTreeLITLEN.resetCurNode()
foundLITLEN = False
# A distance is considered "found" by default in case the character read is
just a literal
distFound = True

# While a literal or length isn't found in the LITLEN tree, keep searching
bit by bit
while(not foundLITLEN):
    curBit = str(self.readBits(1))
    # Updates the current node according the the bit just read
    codeLITLEN = HuffmanTreeLITLEN.nextNode(curBit)
    # If a leaf is reached in the LITLEN tree, follow the instructions
according to the value found
    if (codeLITLEN != -1 and codeLITLEN != -2):
       foundLITLEN = True
```

Quando uma folha é encontrada, é necessário saber se o caracter associado se trata de um literal ou um comprimento (os literais são os valores menores que 256 e os comprimentos são os valores maiores que 256), caso se trate de um literal, basta adicionar ao output o respetivo caracter.

```
# If the code reached is in the interval [0, 256[, just append the value
read corresponding a the literal to the output array
if(codeLITLEN < 256):
    output += [codeLITLEN]
```

Caso se trate de um caracter `c` de comprimento, há algumas regras especiais para saber o seu valor. Se c estiver no intervalo `[257, 265]`, o comprimento é definido por `c - 257 + 3` ou apenas `c - 254`, no entanto, para valores maiores que `265`, será necessário ler bits extra, a quantidade de bits extra a ler é definida por o índice `c - 265` do array `ExtraLITLENBits = [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5]` e o valor do comprimento é definido pela soma mesmo índice do array `ExtraLITLENLens = [11, 13, 15, 17, 19, 23, 27, 31, 35, 43, 51, 59, 67, 83, 99, 115, 131, 163, 195, 227]` e dos extra bits acabados de ler, ou seja: `ExtraLITLENBits[c - 265]+ExtraLITLENLens[c - 265]`.

```
# if the code is in the interval [257, 265[, sets the length of the string
to copy to the code read - 257 + 3
if(codeLITLEN < 265):
   length = codeLITLEN - 257 + 3

# the codes in the interval [265, 285] are special and require more bits to
be read
else:
   # dif defines the indices in the "Extra array's" to be used
   dif = codeLITLEN - 265
   # How many extra bits will need to be read
   readExtra = ExtraLITLENBits[dif]
   # How much extra length to add
   lenExtra = ExtraLITLENLens[dif]
   length = lenExtra + self.readBits(readExtra)
```

Após ser obtido um valor de comprimento, resta saber a distância a recuar, para isso, será feita uma procura na árvore até `HuffmanTreeDIST` ser encontrada uma folha.

```
while(not distFound):
   distBit = str(self.readBits(1))
   # Updates the current node according to the bit just read
   codeDIST = HuffmanTreeDIST.nextNode(distBit)

   # If a leaf is reached in the LITLEN tree, follow the instructions
according to the value found
   if(codeDIST != -1 and codeDIST != -2):
       distFound = True
```

Tal como na leitura dos comprimentos, há algumas regras especiais a aplicar após a leitura do caracter de distância `d`. Se `d` estiver no intervalo [0, 3], o valor da distância é dado por `d + 1`, caso contrário será necessário ler bits extra, a quantidade de bits extra a ler é definida pelo índice `d - 4` do array `ExtraDISTBits = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13]` e o valor da distância é dado pela soma do mesmo índice do array `ExtraDISTLens = [5, 7, 9, 13, 17, 25, 33,`

`49, 65, 97, 129, 193, 257, 385, 513, 769, 1025, 1537, 2049, 3073, 4097, 6145, 8193, 12289, 16385, 24577]`, ou seja, `ExtraDISTBits[d-4] + ExtraDISTLens[d-4]`.

```
# If the code read is in the interval [0, 4[ define the distance to go back to the code read + 1
if(codeDIST < 4):
   distance = codeDIST + 1

# The codes in the interval [4, 29] are special and require more bits to be read
else:
   # dif defines the indices in the "Extra arrays" to be used
   dif = codeDIST - 4
   readExtra = ExtraDISTBits[dif]
   # How many extra bits need to be read
   distExtra = ExtraDISTLens[dif]
   # How much extra distance to add
   distance = distExtra + self.readBits(readExtra)
```

No final deste processo, o comprimento (`length`) e a distância (`distance`) estarão guardados em memória, portanto, resta copiar o valor de `length` valores a partir do índice `-distance` para o início do output. Na eventualidade de `len(output) - distance + length` ser maior que o comprimento atual do output, isto não será um problema e serão copiados os caracteres acabados de ser adicionados.

```
# For each one of the range(length) iterations, copy the character at index len(output)-distance to the end of the output array
for i in range(length):
    output.append(output[-distance])
```

Neste algoritmo, devem ser sempre mantidos em memória os últimos 32768 caracteres lidos, ou seja, é sempre possível recuar uma distância de 32768, mesmo que algumas distâncias estejam em blocos anteriores ao bloco que está atualmente a ser lido. Para esse efeito, foram consideradas duas alternativas:

- A partir do momento em que o array de output em memória chega ao comprimento 32768, sempre que se lê um novo caracter, o caracter há mais tempo no array é removido e escrito no ficheiro .txt, sendo o caracter acabado de ler adicionado ao final do array.
- No 1nal de cada bloco, os caracteres em excesso (`len(output) - 32768`) são escritos no ficheiro .txt e removidos da memória do programa, sendo apenas guardados os últimos 32768.

A segunda opção foi a escolhida, visto que a primeira implica, sempre que um caracter é lido, mover todos os outros elementos do array da memória, algo extremamente

ineficiente. Para a implementação da segunda opção, a seguinte operação acontece sempre após a leitura de um bloco:

```
if(len(output) > 32768):
    # Write every charater that exceeds the 327680 range to the file
    f.write(bytes(output[0 : len(output) - 32768]))
    # Keep the rest in the output array
    output = output[len(output) - 32768 :]
```

Quando todos os blocos tiverem sido lidos, resta guardar no ficheiro .txt os bytes atualmente no array de output.

```
f.write(bytes(output))
```

3 Conclusões

O projeto resultou no desenvolvimento de um decodificador funcional para blocos comprimidos com o uso de códigos de Huffman dinâmicos e o algoritmo *deflate*. Além disso, proporcionou um aprofundamento significativo no entendimento dos princípios por trás dos métodos de compressão de dados.