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 Marques

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 iniciar o processo, é fundamental compreender a estrutura de cada bloco de dados (Imagem 3.1.1). Essa estrutura está dividida em três partes principais:

- Header (Cabeçalho):
  - Composto por apenas 3 bits
    - O primeiro indica se o bloco atual é o último.
    - Os 2 bits seguintes especificam o tipo de compressão utilizado.
  - Como este projeto se concentra exclusivamente na descompressão de blocos comprimidos com árvores de Huffman dinâmicas, os dois últimos bits não serão importantes.
- Códigos Huffman:
  - Essa seção define os códigos de Huffman que serão utilizados no processo de descompressão. Os elementos são definidos da seguinte maneira:
    - HLIT (5 bits): Indica o número de códigos no alfabeto de literais/comprimentos. O valor está no intervalo [257 - 286].
    - HDIST (5 bits): Determina o número de códigos no alfabeto das distâncias, variando entre [1 - 32].

        - HCLEN (4 bits): Define o número de códigos no alfabeto dos comprimentos de código.
    - Após esses 14 bits iniciais, seguem-se (HCLEN + 4) * 3 bits. Cada grupo de 3 bits descreve o comprimento de um código na árvore dos comprimentos de código, correspondente aos elementos da tabela [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15].
        - Em seguida, são definidos HLIT + 257 códigos que representam os comprimentos de código dos elementos na árvore de literais/comprimentos.
        - Por fim, são definidos HDIST + 1 códigos, correspondentes aos comprimentos de código dos elementos na árvore das distâncias.
- Data Bytes (Dados Comprimidos):
    - Nesta parte, encontram-se os dados comprimidos que serão descompactados utilizando as árvores de Huffman configuradas na seção anterior.

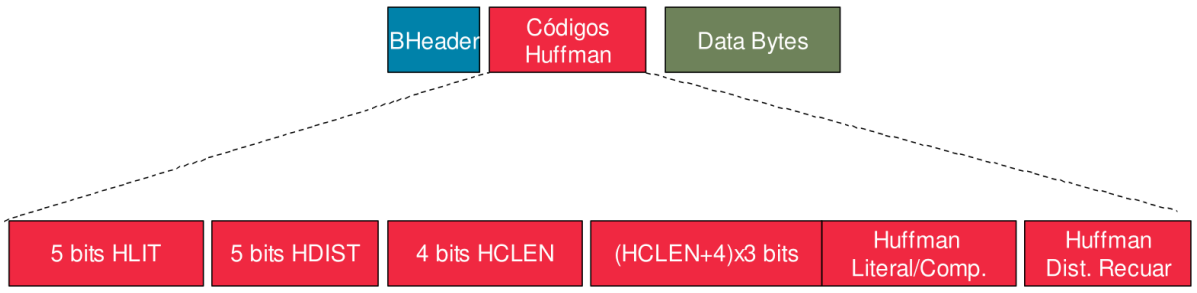

Imagem 3.1.1: Estrutura de um bloco

2.2 Definição dos códigos de Huffman

2.2.1 Leitura de HLIT, HDIST e HCLEN

A etapa inicial consiste, naturalmente, na leitura dos parâmetros *HLIT*, *HDIST* e *HCLEN*. Esse processo é realizado pelo método readDynamicBlock, que utiliza o método auxiliar readBits para interpretar os valores correspondentes:

```
def readDynamicBlock (self):

    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

Na sequência, realiza-se a leitura de (HCLEN + 4) × 3 bits, onde cada grupo de 3 bits define o comprimento do código na árvore de comprimentos de código. A posição de leitura segue a ordem especificada pelos índices da tabela:

[16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15].

Esse processo é implementado pelo método storeCLENLengths. A função interpreta os subconjuntos de 3 bits e associa os comprimentos de código às posições correspondentes na tabela.

```
def storeCLENLengths(self, HCLEN):

  # Ordem dos comprimentos na qual os bits são lidos
  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
  # Alfabeto de comprimentos tem length 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

Considerando que as seções subsequentes do bloco dependem da árvore de Huffman dos comprimentos de código, ela deve ser construída nesta etapa do programa. Para isso, utiliza-se o método createHuffmanFromLens, cujo algoritmo é fundamentado em duas regras principais:

- Códigos com o mesmo comprimento possuem valores consecutivos em ordem lexicográfica. Por exemplo, se os elementos 'C' e 'D' têm comprimento 3 e são os únicos com essa característica, os códigos seriam atribuídos como ‘C’ com 110 e ‘D’ com 111;

- Códigos mais curtos vêm antes dos mais longos na ordem lexicográfica;

Com base nessas regras, o algoritmo é dividido em duas etapas:

Inicialmente, armazena-se no índice *i* do array *bl_count* o número de códigos com comprimento *i*.

```
bl_count = [0 for i in range(max_len+1)]
# Obter um array com o número de códigos de comprimento N (bl_count)
for N in range(1, max_len+1):
    bl_count[N] += lenArray.count(N)
```

A seguir, determinamos o "ponto de partida" para cada comprimento de código, ou seja, o valor numérico correspondente ao menor código possível para cada comprimento. Esse valor inicial para o comprimento N será armazenado no índice N do array *next_code*, conforme o seguinte algoritmo.

```
# Obter primeiro código de cada comprimento de código
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
```

Por fim, atribuímos valores numéricos a todos os códigos, incrementando-os para cada comprimento específico. Começamos com os códigos iniciais definidos no passo anterior, adicionamos cada código gerado à árvore de Huffman *htr* e associamos cada código ao seu respectivo símbolo.

```
# Define códigos para cada símbolo em ordem lexicográfica
for n in range(max_symbol):
    # Comprimento associado ao símbolo n
    length = lenArray[n]
    if(length != 0):
        code = bin(next_code[length])[2:]
        # Caso haja 0s no início do código,
        # temos que adicioná-los manualmente
        # length-len(code) 0s devem ser adicionados
        extension = "0"*(length-len(code))
        htr.addNode(extension + code, n, verbose)
        next_code[length] += 1
```

Assim sendo, o método devolve a árvore htr, e o valor de retorno é guardado na variável HuffmanTreeCLENs.

```
# Com base nos CLEN code lens da árvore, define a huffman tree para CLEN
HuffmanTreeCLENs = self.createHuffmanFromLens(CLENcodeLens, verbose=False)
```

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

As árvores de Huffman do alfabeto de literais/comprimentos e de distâncias serão construídas de forma semelhante à árvore HuffmanTreeCLENs. Contudo, diferentemente desta última, onde os comprimentos de código foram lidos diretamente em grupos de 3 bits, os comprimentos de código das árvores de literais/comprimentos e de distâncias serão representados como códigos na árvore HuffmanTreeCLEN.
O método responsável por esta leitura é o storeTreeCodeLens, que recebe 2 parâmetros: o tamanho da árvore cujos comprimentos de código serão armazenados e também a árvore dos comprimentos de código (HuffmanTreeCLEN).

Como as seções do bloco que contêm os comprimentos de código das árvores de literais/comprimentos e de distâncias não possuem um tamanho fixo, ao contrário da anterior (com tamanho constante de (HCLEN + 4) × 3), a leitura deve continuar até que os arrays de comprimentos de código (treeCodeLens) atinjam os tamanhos esperados: HLIT + 257 para a árvore de Literais/Comprimentos, ou HDIST + 1 para a árvore de distâncias.

Isso significa que o código do método será executado dentro do seguinte loop:

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

Em cada iteração, o nó atual é posicionado na raiz da árvore HuffmanTreeCLEN. Em seguida, os bits são lidos sequencialmente até que uma folha da árvore seja encontrada. Esse processo é realizado por meio do método nextNode da classe HuffmanTree. A verificação baseia-se no comportamento do método: ele retorna -1 quando o código atual é inválido e -2 quando o nó atual não corresponde a uma folha.

```
# Define o nó atual para a raiz da árvore
CLENTree.resetCurNode()
found = False
# Durante a leitura, se uma folha não foi encontrada, continue procurando bit a bit
while(not found):
    curBit = self.readBits(1)
    code = CLENTree.nextNode(str(curBit))
    if(code != -1 and code != -2):
        found = True
```

Ao encontrar uma folha ainda temos que levar os seguintes caracteres especiais em consideração, antes de colocar o símbolo da folha no array:

- 16: Lê mais 2 bits que definem quantas vezes se copiará 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 adicionado ‘0’. O número de adições pode variar entre 3 e 10.
- 18: Lê mais 7 bits que definem quantas vezes será adicionado ‘0’. O número de adições pode variar entre 11 e 138.

Estes caracteres especiais são verificados e tratados da seguinte maneira, conforme indica no algoritmo:

```
if(code == 18):
  amount = self.readBits(7)
  # De acordo com os 7 bits que acabamos de ler, defina os seguintes valores 11-138 no array de comprimento como 0
  treeCodeLens += [0]*(11 + amount)
if(code == 17):
  amount = self.readBits(3)
  # De acordo com os 3 bits que acabamos de ler, defina os seguintes 3-10 valores no array de comprimento como 0
  treeCodeLens += [0]*(3 + amount)
if(code == 16):
  amount = self.readBits(2)
  # De acordo com os 2 bits que acabamos de ler, defina os seguintes 3-6 valores no array de comprimento para o comprimento mais recente lido
  treeCodeLens += [prevCode]*(3 + amount)
elif(code >= 0 and code <= 15):
  # Se um caractere especial não for encontrado, basta definir o próximo comprimento do código para o valor encontrado
  treeCodeLens += [code]
  prevCode = code
```

Assim, o método retorna `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 criadas a seguir de maneira similar a HuffmanTreeCLENs.

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

2.3 Leitura dos dados

Depois que as árvores HuffmanTreeLITLEN e HuffmanTreeDIST estiverem definidas, o próximo passo é ler a parte final do bloco contendo os dados comprimidos. Essa descompressão será realizada conforme o algoritmo LZ77 no método decompressLZ77.

O código do caractere 256 na árvore HuffmanTreeLITLEN representa um caractere especial que sinaliza o final do bloco. Com isso, a leitura dessa parte do bloco será realizada no ciclo while a seguir:

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

Cada ciclo começa por adicionar o nó atual na raiz da árvore HuffmanTreeLITLEN, e ler bits sequencialmente até encontrar uma folha dessa árvore.

```
# Redefine o nó atual para a raiz da árvore
HuffmanTreeLITLEN.resetCurNode()
foundLITLEN = False
# Uma distância é considerada "encontrada" por padrão caso o caractere lido seja apenas um literal
distFound = True

# Embora um literal ou comprimento não seja encontrado na árvore LITLEN, continue pesquisando bit a bit
while(not foundLITLEN):
   curBit = str(self.readBits(1))
   # Atualiza o nó atual de acordo com o bit que acabou de ler
   codeLITLEN = HuffmanTreeLITLEN.nextNode(curBit)
   # Caso seja atingida uma folha na árvore LITLEN, siga as instruções de acordo com o valor encontrado
   if (codeLITLEN != -1 and codeLITLEN != -2):
      foundLITLEN = True
```

Ao encontrar uma folha, é preciso determinar se o caractere associado é um literal ou um comprimento (os literais são valores menores que 256, enquanto os comprimentos são valores maiores que 256). Se for um literal, basta adicionar o caractere correspondente ao output.

```
# Se o código alcançado estiver no intervalo [0, 256[, basta anexar o valor lido correspondente ao literal ao array de saída
if(codeLITLEN < 256):
   output += [codeLITLEN]
```

Caso seja um caracter de comprimento `c`, há algumas regras para saber o seu valor. Se c se encontrar dentro do intervalo [257, 265], o comprimento é definido por `c - 257 + 3` (ou `- 254`). Ainda assim, para valores maiores que 265, será preciso ler bits extras, e sua quantidade a ler é definida pelo 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 do 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 lidos, ou seja:

ExtraLITLENBits[c - 265]+ExtraLITLENLens[c - 265].

```
# se o código estiver no intervalo [257, 265[, define o comprimento da string a ser copiada para o código lido - 257 + 3
if(codeLITLEN < 265):
  length = codeLITLEN - 257 + 3

# os códigos no intervalo [265, 285] são especiais e requerem mais bits para serem lidos
else:
  # dif define os índices nos "arrays extras" a serem usados
  dif = codeLITLEN - 265
  # Quantos bits extras precisarão ser lidos
  readExtra = ExtraLITLENBits[dif]
  # Quanto comprimento extra adicionar
  lenExtra = ExtraLITLENLens[dif]
  length = lenExtra + self.readBits(readExtra)
```

Depois de se obter um valor de comprimento, é preciso determinar a distância a recuar. Para isso realiza-se uma busca na árvore até encontrar uma folha em HuffmanTreeDIST.

```
while(not distFound):
  distBit = str(self.readBits(1))
  # Atualiza o nó atual de acordo com o bit que acabou de ler
  codeDIST = HuffmanTreeDIST.nextNode(distBit)

  # Caso seja atingida uma folha na árvore LITLEN, siga as instruções de acordo com o valor encontrado
  if(codeDIST != -1 and codeDIST != -2):
    distFound = True
```

Assim como na leitura dos comprimentos, existem algumas regras especiais a serem seguidas após a leitura do caractere de distância d. Se d estiver no intervalo [0, 3], o valor da distância será d + 1. Caso contrário, será necessário ler bits adicionais, e a quantidade de bits extras a serem lidos é definida pelo índice d - 4 no 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]`.

O valor da distância será então a soma do valor correspondente desse índice no 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]`.

```
# Se a leitura do código estiver no intervalo [0, 4[ defina a distância para voltar até a leitura do código + 1
if(codeDIST < 4):
  distance = codeDIST + 1

# Os códigos no intervalo [4, 29] são especiais e requerem mais bits para serem lidos
else:
  # dif define os índices nos "Arrays extras" a serem usados
```

```
dif = codeDIST - 4
readExtra = ExtraDISTBits[dif]
# Quantos bits extras precisam ser lidos
distExtra = ExtraDISTLens[dif]
# Quanta distância extra adicionar
distance = distExtra + self.readBits(readExtra)
```

Ao final desse processo, os valores de comprimento (length) e distância (distance) estarão armazenados em memória. Portanto, o próximo passo é copiar os valores de comprimento a partir do índice -distance para o início do output. Caso a expressão len(output) - distance + length seja maior que o tamanho atual do output, isso não será um problema, e os caracteres recém-adicionados serão copiados conforme necessário.

```
# Para cada uma das iterações de intervalo (comprimento), copie o caractere no índice
len(output)-distance para o final da matriz de saída
for i in range(length):
    output.append(output[-distance])
```

Neste algoritmo, é necessário manter em memória os últimos 32768 caracteres lidos, o que garante a possibilidade de recuar uma distância de até 32768, mesmo que algumas distâncias se refiram a blocos anteriores ao bloco atualmente em leitura. Para isso, foram consideradas duas alternativas:

- A primeira alternativa envolve, sempre que o array de output atingir 32768 caracteres, a remoção do caractere mais antigo no array e sua escrita no arquivo .txt, com o novo caractere lido sendo adicionado ao final do array.
- Na segunda opção, ao final de cada bloco, os caracteres excedentes (len(output) - 32768) são escritos no arquivo .txt e removidos da memória, mantendo apenas os últimos 32768.

A segunda alternativa foi a escolhida, pois a primeira exigiria, a cada leitura de um caractere, a movimentação de todos os outros elementos no array da memória, o que seria extremamente ineficiente. Para a implementação dessa opção, a seguinte operação ocorre sempre após a leitura de um bloco:

```
if(len(output) > 32768):
    # Grave todos os caracteres que excedam o intervalo 327680 no arquivo
    f.write(bytes(output[0 : len(output) - 32768]))
    #  Mantenha o resto no array de saída
    output = output[len(output) - 32768 :]
```

Após a leitura de todos os blocos, basta guardar no ficheiro .txt os bytes atuais do array de output.

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

3 Conclusões

Ao finalizar este projeto, foi desenvolvido um decodificador de blocos comprimidos com códigos de Huffman dinâmicos utilizando o algoritmo *deflate*, além de proporcionar um entendimento mais aprofundado sobre algoritmos de compressão de dados.