1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA

dei departamento de engenharia informática

RELATÓRIO

TP2 - Descompactação de ficheiros 'gzip'

Unidade Curricular:

Teoria da Informação

Data:

16/12/2023

João Nuno Coelho
No 2021275030

Luana Carolina Reis
No 2022220606

Rodrigo Barata
No 2022212664

INTRODUÇÃO

Com a realização deste trabalho prático, pretende-se implementar um descodificador do algoritmo *deflate*, utilizado em ficheiros do tipo ".gzip". Esta implementação servirá para descompactar blocos de informação comprimidos com recurso a códigos de Huffman dinâmicos (BTYPE = 10).

O código será testado através de diversos ficheiros de exemplo, que contêm um número variado de blocos a descomprimir. Serão, portanto, implementadas medidas que permitam lidar com a leitura de múltiplos blocos.

Estrutura do código

É nos fornecido um ficheiro "huffmantree.py", que contém uma classe "HuffmanTree", responsável por criar, aceder e gerir árvores de Huffman e um ficheiro "gzip.py", que lê e armazena o header do ficheiro, através da classe "GZIPHeader".

Uma vez que estas questões já estão tratadas, o nosso foco será na classe "GZIP", a classe responsável pela descompactação do ficheiro ".gz", bloco a bloco.

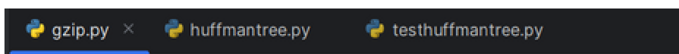

Fig. 1 - Ficheiros incluídos no projeto

Algoritmo *deflate*, dicionário LZ77 e Huffman *coding*

O algoritmo de compressão *deflate* tem como objetivo principal reduzir o tamanho dos dados, sem perder qualquer informação, e utiliza uma combinação de outros dois algoritmos: Lempel-Ziv (LZ77) e Huffman *coding*.

O dicionário LZ77 é uma estrutura de dados que armazena sequências de dados com padrões repetitivos. Quando uma certa sequência é encontrada novamente, em vez de voltar a armazená-la, o algoritmo faz referência à sua posição no dicionário, economizando-se assim espaço de armazenamento.

Já o Huffman coding, tem como objetivo atribuir códigos de comprimento variável aos diferentes símbolos, de maneira a que os símbolos mais frequentes recebam códigos mais curtos, reduzindo-se o espaço necessário para a representação da informação.

ESTRUTURA DE UM BLOCO

a) BHeader

O “BHeader“ é composto por três campos. Para BFINAL é reservado um bit. Caso seja igual a 1, estamos no último bloco e caso contrário, ainda temos blocos a percorrer.

Para BTYPE são reservados dois bits, que definem o tipo de compressão utilizada (00, 01, 10, 11). Para este trabalho, só nos é relevante o penúltimo caso.

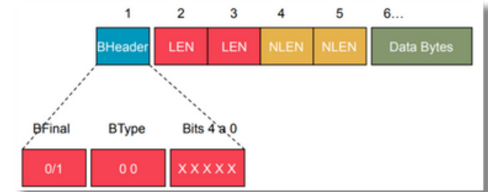

Fig. 2 – Componentes de “BHeader“

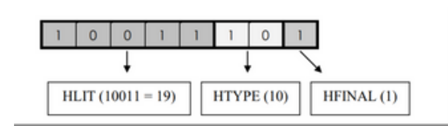

Fig. 3 – *Block header*

b) Códigos Huffman

HLIT: # de códigos literais/comprimento – 257 (257 – 286)
HDIST: # de códigos de distância – 1 (1 – 32)
HCLEN: # de códigos de “comprimento de código“ – 4 (4 – 19)

(HCLEN + 4) x 3: sequência de 3 bits com os comprimentos dos códigos do “alfabeto do comprimento de códigos“, pela seguinte ordem:

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

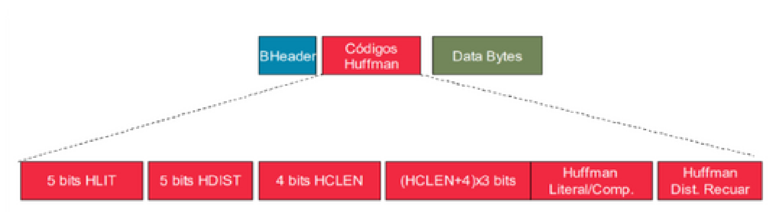

Fig. 4 – Componentes de “Códigos Huffman“

Símbolos:

0-255: literal
256: fim de bloco
257-285: <comprimento, distância a recuar>

Fig. 5 – Interpretação do valor dos símbolos

c) Data Bytes

A secção “Data Bytes“ refere-se aos dados comprimidos concretos (num determinado bloco), armazenados sob a forma de *bytes*.

É efetuado um loop até ao último bloco e, de acordo com o valor de literal/comprimento lido a partir da “input stream“, é efetuada uma determinada ação.

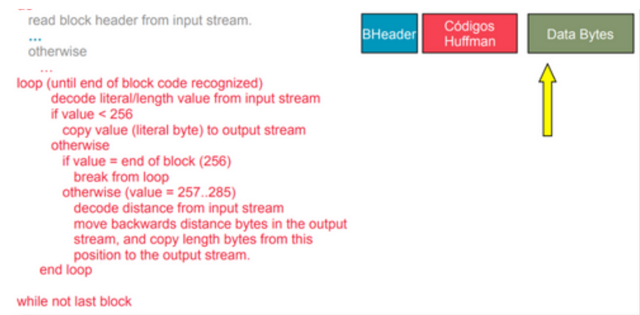

Fig. 6 – Loop sobre a informação

DESCOMPRESSÃO

1) Leitura do formato do bloco (HLIT, HDIST e HCLEN)

No início de cada bloco, são lidos:

- 5 bits para o valor de HLIT
- 5 bits para o valor de HDIST
- 4 bits para o valor de HCLEN.

A HLIT será adicionado 257, de forma a obter o número de nós da árvore de literais/comprimentos.
A HDIST será adicionado 1, de forma a obter o número de nós da árvore de distâncias.
A HCLEN será adicionado 4, de forma a obter o número de nós da árvore de "comprimentos de códigos".

2) Armazenamento dos comprimentos dos códigos do "alfabeto de comprimento de códigos", com base em HCLEN

Após a leitura do formato do bloco, serão lidos 3 bits, HCLEN vezes.
Estes 3 bits dão informação sobre o tamanho do código para alcançar cada nó na árvore, segundo uma ordem específica.

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

Isto é, os primeiros 3 bits lidos dizem-nos o tamanho necessário para o código do nó 16, os próximos 3 bits dizem o tamanho para o código do nó 17 e assim sucessivamente.
Note-se que a leitura não termina necessariamente no último número desta ordem, mas sim no HCLEN'ésimo número desta ordem.
Além disso, também é guardado numa lista o valor de nós que precisam do mesmo número de bits, ou seja, uma lista de ocorrências de bits necessários para todos os códigos ("hclen_array").

```
HCLEN Array:  [3, 0, 0, 5, 4, 4, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 5, 5, 5]
```

Fig. 7 - "hclen_array' para o ficheiro "FAQ.txt.gz"

DESCOMPRESSÃO

3) Conversão dos comprimentos dos códigos do ponto anterior em códigos de Huffman do "alfabeto de comprimentos de códigos"

Sabendo o tamanho de bits para codificar cada nó (bit_count) e as ocorrências dos tamanhos do códigos (hclen_array), conseguimos obter os códigos de cada nó. Para isto, criámos a função "cria_codigos', seguindo o algoritmo dos slides.

```
Code – código base
bl_count [i] – número de símbolos a serem codificados com i bits

code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}

for (n = 0;  n <= max_code; n++) {
    len = tree[n].Len;
    if (len != 0) {
        tree[n].Code = next_code[len];
        next_code[len]++;
    }
  }
```

Fig. 8 – Algoritmo dos 2 ciclos for (slides)

4) Ler e armazenar os HLIT + 257 comprimentos dos códigos referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman de comprimentos de códigos

- Recorrer a funções do ficheiro "huffmantree.py', nomeadamente às funções de pesquisa de códigos de forma sequencial (bit a bit);
- Alguns códigos requerem a leitura de alguns bits extra (índices 16, 17 e 18 do alfabeto).

Através de uma função "ler_comprimentos" é lido um número ("tamanho" = HLIT + 257) de comprimentos de códigos, a partir de uma árvore de Huffman ("hclen_tree").

O objetivo é construir a lista "symbols", que armazena os comprimentos dos códigos dos valores para a "hlit_tree".

Dentro de um loop, é lido um bit usado para navegar na árvore de Huffman e encontrar um "nó". O valor do índice do nó encontrado é então verificado para saber que ação tomar.

Se o valor do índice estiver entre 0 e 15, é adicionado diretamente à lista de símbolos. Se for 16, repete-se o último valor lido, "rep" vezes. Se for 17 ou 18, adiciona-se um certo valor de zeros, "rep" vezes. (Fig. 9)

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```

Fig. 9 – Casos especiais (índices 16, 17 e 18)

DESCOMPRESSÃO

5) Ler e armazenar os HDIST + 1 comprimentos de código referentes ao alfabeto de literais/comprimentos, codificados segundo o código de Huffman de comprimentos de códigos

- Recorrer a funções do ficheiro "huffmantree.py', nomeadamente às funções de pesquisa de códigos de forma sequencial (bit a bit);
- Alguns códigos requerem a leitura de alguns bits extra (índices 16, 17 e 18 do alfabeto).

Através de uma função "ler_comprimentos" é lido um número ("tamanho" = HDIST + 1) de comprimentos de códigos, a partir de uma árvore de Huffman ("hclen_tree").

Durante o seu funcionamento, é lido 1 bit dentro de um ciclo "while" e, dependendo do seu valor, a "hclen_tree" é percorrida até se encontrar uma folha. O valor do índice nesta será determinante sobre como proceder.

Desta vez, a lista "symbols" será composta dos comprimentos dos códigos dos valores para a "hdist" .

Da mesma forma que a árvore anterior, se o valor do índice estiver entre 0 e 15, é adicionado diretamente à lista de símbolos. Caso seja 16, o último valor lido é repetido "rep" vezes. Por fim, se for 17 ou 18, adiciona-se um certo valor de zeros, "rep" vezes.

6) Determinar e armazenar os códigos de Huffman referentes aos dois alfabetos, literais/comprimentos e distâncias

- Usar o método do ponto 3 (converter os comprimentos dos códigos em códigos de Huffman do "alfabeto de comprimentos de códigos').

Neste ponto, pretendem-se criar códigos de Huffman para HLIT e para HDIST. Para cada um destes casos, é criada uma lista de contagem de repetições para os símbolos e, de seguida, os códigos de Huffman são gerados, usando o método "cria_codigos", sendo posteriormente convertidos para strings, através do método "to_string".

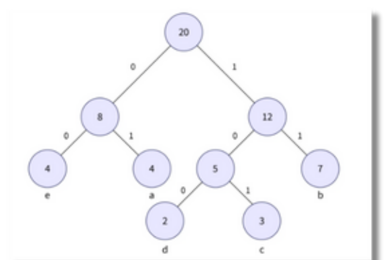

Fig. 10 - Huffman *coding*

DESCOMPRESSÃO

7) Criar as funções necessárias à descompactação dos dados comprimidos, com base nos códigos de Huffman e no algoritmo LZ77

- Recorrer a funções do ficheiro “huffmantree.py’, nomeadamente às funções de pesquisa de códigos de forma sequencial (bit a bit);
- Alguns códigos requerem a leitura de alguns bits extra:
  índices 265 a 285 no alfabeto de literais/comprimentos
  índices 4 a 29 no alfabeto de distâncias

A partir de duas árvores de Huffman (“hlit_tree“ e “hdist_tree“), será então começada a descompactação dos dados.
Dentro de um ciclo while, “hlit_tree“ é lida nó a nó, usando o método “nextNode“ do ficheiro “huffmantree.py“ e armazenado o valor devolvido. Quando é atingida uma folha, o valor final será tratado, segundo os seguintes pontos:

- <u>Menor ou igual a 255</u>: o valor é copiado diretamente para a lista “output “;
- <u>Igual a 256:</u> finaliza o ciclo, indicando o fim do bloco;
- <u>Para valores de 257 a 285</u>: são definidos 2 parâmetros, uma distância a recuar “dist“ e um comprimento a copiar “length“.

O primeiro é obtido fazendo uma leitura de “hdist_tree“, enquanto o segundo provém de um cálculo em que intervêm o valor lido de “hlit_tree“ e o valor obtido de um certo número de bits lidos do gzip (257-264 e 285: nada é lido, 265-268: lê-se 1 , 269-272: lê-se 2 , 273-276: lê-se 3 , 277-280: lê-se 4 e de 281-284: lê-se 5), de modo a centrá-lo na faixa de número de símbolos a copiar condizente com a faixa de valores em que se enquadra o valor lido (Figs. 12 e 13).

Após definidos, “length“ e “dist“, em conjunto com uma lista “output“, são passados como parâmetros para uma função recursiva “alterar_output“, que executa a “essência“ do LZ77, ou seja, recua o ponteiro “dist“ símbolos e copia os próximos “length“ símbolos à sua frente. No caso de “length“ < “dist“, é adicionado à lista “output“ output[-dist: -dist + length]. Caso contrário, adiciona-se output[-dist] (a função finaliza aqui, se “length“ e “dist“ forem iguais) e, no caso de “length“ > “dist“, é chamada novamente a função “alterar_output“, desta vez com “length“ = “length“ - “dist“ .

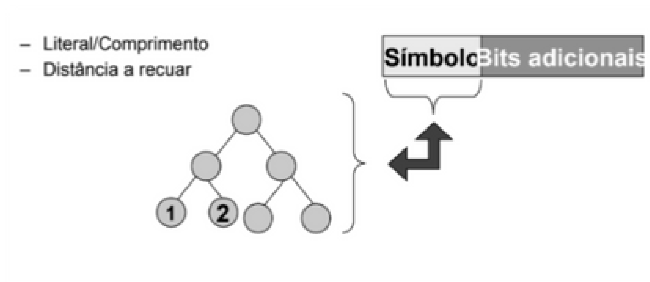

Fig. 11 - Par < comprimento, distância a recuar >

DESCOMPRESSÃO

Code	Extra Bits	Length(s)	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Fig. 12 - Casos especiais (índices 265 - 285)

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Fig. 13 - Casos especiais (índices 4 - 29)

8) Gravar os dados descompactados num ficheiro com o nome original

Antes da leitura de todos os blocos, um ficheiro novo é aberto. O nome deste é igual ao do ficheiro que estamos a descomprimir, mas sem os últimos 3 caracteres (".gz"). De seguida, é criada uma lista ("output"), onde serão guardados os bytes do ficheiro descomprimido.

Para cada bloco, após a leitura e criação de todas as árvores da Huffman, serão lidos vários bits, que serão interpretados segundo estas árvores. Sempre que um *byte* é lido, este é introduzido na lista "output". Sempre que esta lista passa a ter um tamanho superior a 50 000 *bytes*, os primeiros 10 000 *bytes* serão escritos no ficheiro aberto e apagados da lista. Estes valores foram escolhidos consoante o limite de leitura de 2^15 = 32 768 *bytes*, ou seja, é necessário ler um número de bytes inferior a este (10 000, neste caso), sendo que o 50 000 age como limitador superior dos mesmos.

Após a leitura de todos os blocos, os *bytes* restantes no "output" serão escritos no ficheiro e este será fechado.

```
# Mantém-se o ficheiro aberto até ao final da leitura total do ficheiro
file = open(self.gzFile[:-3], "wb")

# O 'output' é inicializado aqui por ter de começar desde o início do ficheiro, e não desde o início de cada bloco
output = []
```

Fig. 14 - Abertura do ficheiro em modo "write binary"

CONCLUSÃO

Após o desenvolvimento do algoritmo de descompressão de blocos de informação, foram testadas as funcionalidades implementadas em diferentes tipos de ficheiros, nomeadamente em arquivos ".jpeg" (imagem), ".txt" (texto) e ".mp3" (áudio) e obtivemos os ficheiros descomprimidos com sucesso.

Resultados do algoritmo de descompressão

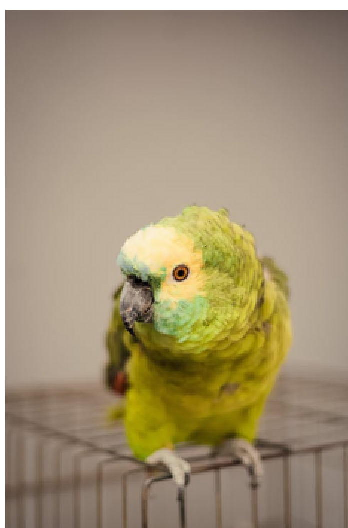

Fig. 16 – Imagem 'sample_image.jpeg'

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut

In nisl nisi scelerisque eu ultrices vitae auctor eu. Dolor sit amet consectetur adipiscing e

Urna nunc id cursus metus. Id leo in vitae turpis massa. Blandit turpis cursus in hac habitas

Egestas egestas fringilla phasellus faucibus scelerisque. Sit amet dictum sit amet justo done

Laoreet suspendisse interdum consectetur libero id faucibus nisl tincidunt. Enim sit amet ven

Nibh praesent tristique magna sit amet purus gravida quis. Commodo viverra maecenas accumsan

Eget mi proin sed libero enim sed faucibus turpis in. Sed odio morbi quis commodo odio aenean

Nunc lobortis mattis aliquam faucibus purus in massa tempor. Amet consectetur adipiscing elit
```

Fig. 15 – Ficheiro de texto 'sample_large_text.txt'

```

        Frequently Asked Questions about zlib


If your question is not there, please check the zlib home page
http://www.cdrom.com/pub/infozip/zlib/ which may have more recent information.


1) I need a Windows DLL
2) I need a Visual Basic interface to zlib
3) compress() returns Z_BUF_ERROR
4) deflate or inflate returns Z_BUF_ERROR
5) Where is the zlib documentation (man pages, etc...)?
6) Why don't you use GNU autoconf, libtool, etc...?
7) There is a bug in zlib.
8) I get "undefined reference to gzputc"
```

Fig. 17 – Ficheiro de texto 'FAQ.txt'

GZIP

WHAT? WHY? HOW?

REFERÊNCIAS BIBLIOGRÁFICAS

[1] CARVALHO, Paulo. *Trabalho Prático Número 2 - Deflate*. [PDF]. [Consultado em 06/11/2023].

[2] DEUTSCH, Peter. *DEFLATE Compressed Data Format Specification version 1.3*. [PDF]. [Consultado em 06/11/2023].

[3] DEUTSCH, Peter. *GZIP file format specification version 4.3*. [PDF]. [Consultado em 06/11/2023].