UNIVERSIDADE D COIMBRA

15-12-2022

Descompactação de Ficheiros GZIP

Trabalho Prático nº2

Ana Carolina Morais	2021222056
Fernanda Fernandes	2021216620
Inês Quintal	2021232993

Índice

Introdução 2
Exercício 1: *readBits ()* 2
Exercício 2: 3
Exercício 3: 3
Exercício 4: 4
Exercício 5: 5
Exercício 6: Código *Huffman* literais/distâncias 5
Exercício 7: Descompactação 6
Exercício 8: Ficheiro 7
Conclusão: 8
Referências: 8

Introdução

O trabalho prático tem como principal objetivo adquirir sensibilidade para as questões fundamentais de teoria de informação, em particular a codificação usando árvores de *Huffman* e dicionários LZ77.

Como tal, os conceitos fundamentais de teoria da informação vão ser aplicados, através da elaboração de um descodificador do algoritmo *deflate*, usado em ficheiros *gzip*.

Para o desenvolvimento do código, utilizámos o ide Spyder (versão 4.2.5). Através do mesmo garantimos que todo o código cumpre os requisitos associados ao *Python*. Uma vez que foi fornecido um ficheiro *gzip.py* onde já estava implementado algum código inicial, é nesse mesmo ficheiro que vamos continuar a desenvolver o nosso código. Para isso, decidimos implementar algumas bibliotecas que não estavam descritas inicialmente, tais como: *Numpy*.

Ao longo deste trabalho são utilizados alguns conceitos teóricos, que serão mais tarde abordados, tais como: *Codificação Lempel-ZIV*, *LZ77* e árvores de *Huffman*.

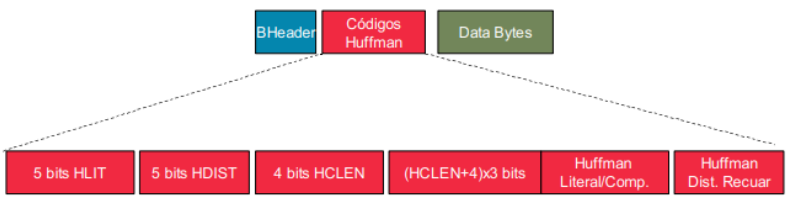
É importante referir que para cada exercício, exceto o primeiro, implementámos um método, uma vez que é mais fácil aceder e é de maior compreensão.

Formato do ficheiro gzip:

Um ficheiro Gzip consiste numa série de blocos de dados comprimidos. Os blocos de dados aparecem seguidos, ou seja, uns atrás dos outros, sem qualquer informação adicional, antes, no meio ou depois dos blocos.

Exercício 1: *readBits ()*

Neste exercício, utilizamos a função *readBits (),* que nos devolve o número de bits do que queremos ler.



Observando a imagem anterior, podemos concluir que 5 bit corresponde ao HLIT, 5 bits ao HDIST e 4 bits ao HCLEN, daí usarmos esses valores para a função *readBits ()*, pois estamos a limitar o número de bits que corresponde a cada parâmetro.

Resultado:

```
HTIL:19
HDIST:22
HCLEN:10
```

Exercício 2:

Para resolver este exercício, começamos por considerar (HCLEN + 4) * 3, que corresponde ás sequencias de 3 bits com os comprimentos dos códigos do alfabeto do "comprimento de códigos" pela seguinte ordem:

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

A partir desta informação, lendo os 3 bits com a função readBits (), conseguimos criar um *array* com os comprimentos dos códigos de acordo com a ordem.

Resultado: n_array = [3,0,0,5,4,4,3,3,3,3,3,0,0,0,0,0,5,5,5]

Exercício 3:

O objetivo deste exercício é converter os comprimentos dos códigos, obtidos no exercício 2 em códigos de *Huffman*. Podemos definir o código de Huffman para um alfabeto fornecendo apenas os comprimentos de bits do código para cada símbolo do alfabeto por ordem, isso é o suficiente para determinar os códigos reais.

Com base no algoritmo dado no documento fornecido (DOC 2), que gera os códigos como inteiros destinados a serem lidos do bit mais significativo para o menos significativo. Por isso, dado esta base, conseguimos implementar o nosso código, começando por contar o número de códigos para cada comprimento de código, de seguida encontramos o valor numérico do menor código para cada comprimento de código.

Por último, convertemos os códigos em binário, de forma a que códigos com comprimento zero não deva ser atribuído nenhum valor, daí a serem uma string vazia, pois se fosse um número inteiro, por exemplo 0, teríamos uma lista heterogénea (com strings e inteiros), o que poderia vir a ser uma dificuldade no futuro.

Resultado:

```
['000', '', '', '11100', '1100', '1101', '001', '010', '011', '100', '101', '', '', '', '', '', '11101', '11110', '11111']
```

Exercício 4:

No início deste exercício começámos por inserir os códigos de Huffman, obtidos no exercício anterior, na nossa árvore, com o método *inserirTree ()*, é de realçar que não inserimos as 'strings' vazias. De seguida, adicionámos 257 ao HLIT, que corresponde aos comprimentos de códigos literais. Após isto, fomos ler os bits usando a função *readBits ()*.

Seguidamente verificámos se o resultado da função está nos comprimentos de códigos obtidos na alínea 3, caso esteja e for um caso especial, ou seja, se o elemento do comprimento de códigos na posição do valor do readBits () for 16/17/18, estamos perante três exceções e, temos de tratar cada uma de forma individualizada.

Sendo assim, quando o valor é igual a 16 iremos ler os dois bits imediatamente a seguir, que nos indica quantas vezes iremos repetir o comprimento do código anterior, o intervalo de repetição é [3,6].

Caso o valor seja igual a 17, iremos ler os três bits imediatamente a seguir e iremos repetir esse número de vezes mais três códigos de comprimento '0', dado que o intervalo de repetições é [3,10].

Por último, caso o valor seja igual a 18, vamos ler os 7 bits imediatamente a seguir e iremos repetir esse número de vezes mais onze códigos de comprimento '0', pois o intervalo de repetições é [11,138].

O resultado obtido corresponde à coluna b do cod. Literal, comp do DOC 5.

Exercício 5:

Para este exercício, implementámos o mesmo método do exercício 4, com a exceção de que agora estamos a tratar o HDIST + 1, que corresponde aos comprimentos dos códigos da distância do alfabeto.

Resultado:

```
[8, 9, 8, 0, 8, 0, 6, 6, 5, 6, 4, 4, 4, 4, 3, 4, 3, 4, 3, 4, 4, 5, 9]
```

Exercício 6: Código *Huffman* literais/distâncias

Para cumprimos o objetivo deste exercício, utilizámos o método criado no exercício 3 para converter os códigos obtidos nas duas alíneas anteriores em códigos de *Huffman*.

Resultado:

1. Código *Huffman* dos literais:

```
['', '', '', '', '', '', '', '', '', '111100010', '1100110', '', '', '1100111', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '0000', '', '1101000', '', '',
'', '', '111100011', '11100110', '1101001', '1111101010', '1111101011', '1101010',
'11100111', '101000', '101001', '1111101100', '11101000', '111100100', '1111101101',
'111100101', '1111101110', '111100110', '111100111', '1111101111', '', '111101000', '', '',
'', '', '1111110000', '111101001', '1111110001', '11101001', '', '11101010', '111101010',
'111101011', '1111110010', '', '111101100', '', '', '11101011', '1111110011', '1111110100',
'1111110101', '1111110110', '1111110111', '11101100', '1111111000', '1111111001',
'111101101', '111101110', '111101111', '1111111010', '', '1111111011', '', '', '', '',
'11101101', '', '0001', '101010', '01010', '101011', '0010', '1101011', '101100', '101101',
'01011', '111110000', '1101100', '101110', '01100', '01101', '01110', '101111', '111110001',
'01111', '10000', '10001', '10010', '1101101', '1101110', '11101110', '1101111', '1110000',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '1111111100',
'0011', '0100', '10011', '110000', '110001', '110010', '1110001', '11101111', '1110010',
'1111111101', '', '1111111110', '1111111111', '', '111110010', '', '111110011', '11110000',
'111110100']
```

2. Código *Huffman* da distância:

```
['11111100', '111111110', '11111101', '', '11111110', '', '111100', '111101', '11100',
'111110', '0110', '0111', '1000', '1001', '000', '1010', '001', '1011', '010', '1100',
'1101', '11101', '111111111']
```

<u>**Nota:**</u> Estes códigos estão no DOC 5.

Exercício 7: Descompactação

De acordo com o Doc 1 e 2, para descompactar os dados comprimidos é importante seguir alguns aspectos, tais como:

```
loop (until end of block code recognized)
   decode literal/length value from input stream
   if value < 256
      copy value (literal byte) to output stream
   otherwise
      if value = end of block (256)
         break from loop
      otherwise (value = 257..285)
         decode distance from input stream

         move backwards distance bytes in the output
         stream, and copy length bytes from this
         position to the output stream.
end loop
```

Assim, numa primeira fase inicial, criámos duas variáveis que vão guardando o número de bits, após a função *readBits ()*, uma guarda os bits literais e a outra os bits da distância a recuar. Também criámos um vetor que vai guardando o número corresponde ao nó e, ainda uma variável que vai conter o texto descodificado.

Para entrar no ciclo descrito acima, realizámos, primeiro, uma procura em cada árvore das variáveis, inicialmente, inicializadas a vazio, respetivamente. Como o código vazio não está introduzido na árvore, o resultado desta operação é -2, que vai ser útil no ciclo para sabermos quando um código chegou à folha certa.

Assim, enquanto o código literal (resultado da função *findNode ()*) for diferente de 256 e, enquanto o código literal for diferente de -2, iremos continuar a procurar na nossa arvore até chegarmos a uma folha, assim que obtemos o número da folha, iremos averiguar se este é menor que 256, maior ou igual.

Caso seja igual, pois é o caso mais fácil, realiza- se um 'break'.

Caso seja inferior, iremos adicionar esse número ao vetor e iremos concatenar à variável designada o mesmo transformado através da função *chr,* uma vez que os carateres estão em ASCII.

Caso seja superior, é necessário recorrer à tabela do slide 15 do Doc 1 para obtermos o comprimento da 'string 'que vamos acrescentar. Uma forma de não recorrermos a tantas condições foi agrupar e encontrar uma fórmula de fácil compreensão.

Por exemplo: quando o código literal varia entre 265 e 268, e, uma vez que todos os elementos que compõem esse intervalo têm o mesmo número de bits extra a ler, este foi um dos critérios que nos levou a esta escolha, depois foi averiguar qual o balanço de um número em relação ao outro.

Nota: Relativamente ao cálculo do comprimento as condições vão só ate ao número 276, uma vez que é o número de elementos presentes no vetor determinado no ponto 4.

De seguida faz-se a leitura de um código presente na árvore de distância e, tendo em conta a tabela do slide 16 do Doc 1, calculamos a distância a recuar, uma vez que no *deflate* ou se adiciona um carater à string ou se recua 'k' elementos e copia 'x' elementos para o final da string. A estrutura deste conceito é <recuar, comp>.

Nota: Relativamente ao cálculo da distância a recuar as condições vão só ate ao número 23, uma vez que é o número de elementos presentes no vetor determinado no ponto 5.

No final, realiza- se um ciclo que repete 'comp' vezes e, neste adiciona-se ao elemento no recuar posição, a contar do fim.

Após a realização do ciclo descrito é determinado a 'string' com o texto descompactado.

Exercício 8: Ficheiro

Por último iremos guardar o conteúdo da 'string' determinada na questão anterior, num ficheiro com o mesmo nome através da utilização da estrutura *GZIPHeader.fName*.

Conclusão:

Em suma, após a criação deste projeto podemos afirmar que conseguimos adquirir melhor sensibilidade a respeito da teoria da informação. Apesar de termos encontrado alguns percalços, sentimos que ultrapassámos as nossas dificuldades. Após uma análise dos resultados, podemos garantir que o objetivo foi comprido.

Referências:

- Doc1 – Slides;
- Doc2 – deflate.rfc;
- Doc3 – gzip header – rfc1952
- Doc5 – Codigos
- https://www.delftstack.com/pt/bincount
- https://docs.python.org/pt-br/3/library/functions.html