FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Teoria da Informação

Trabalho Prático nº2

Professor: Lorena Itatí Petrella

Decode Deflate

João Emanuel Sousa Moreira, 2020230563

Miguel Ângelo Ferreira Miranda, 2021212100

Rodrigo Oliveira de Sá, 2021213188

Licenciatura em Engenharia Informática

2022/2023

Índice

Introdução

Este trabalho tem como objetivo a aprendizagem de questões fundamentais de teoria de informação, em particular, descompactação de blocos comprimidos com códigos de Huffman dinâmicos.

Neste trabalho pretende-se implementar o decodificador do algoritmo deflate.

Desta forma, vamos aplicar este conceito em um ficheiro no formato gzip.

Para a realização do trabalho é fornecido código fonte como base do trabalho.

- Ficheiro gzip.py: Classe principal para a descompactação de um ficheiro no formato gzip.
    - *Classes principais:*
        - **GZIPHEADER:** Classe relativa à leitura do cabeçalho do ficheiro .gz
        - **GZIP:** Classe relativa à descompressão do ficheiro .gz, passando por ler primeiro o header e posteriormente passando para a descompressão bloco a bloco

- Ficheiro huffmantree.py: Classe que contém a classe HuffmanTree com um conjunto de funções para a criação, acesso e gestão de árvores de Huffman.

    - *Classes principais:*
        - **HFNODE:** Contém informação relativa a um nó da árvore de Huffman
        - **HuffmanTree:** Define uma árvore de Huffman

    - *Funções principais:*
        - **addNode:** Adiciona o nó à árvore
        - **nextNode:** Atualiza o nó corrente na árvore com base no nó atual e no próximo bit

- **findNode:** Procura código na árvore, a partir de um nó especificado
- **resetCurNode:** Reposiciona curNode na raíz da árvore

Exercício 1

Para realizar a leitura do hlit, do hdist e do hclen servimo-nos da função readBits. Aqui lemos 5 bits, 5 bits e 4 bits respetivamente.

Exercício 2

Criamos uma lista para os comprimentos dos códigos com base no *HCLEN*, inicializada a zero (este array, funciona como um *hashmap*) e uma lista com a ordem que os comprimentos aparecem [Dados do enunciado].

Lêmos três bits a cada iteração e colocamos no nosso *hashmap.* A posição em que colocamos está de acordo com o array dado no enunciado.

Exercício 3

Criamos uma função *bounds* com o objetivo de calcular o mínimo e máximo do array de comprimento de códigos.

Criamos uma função *ocurrencies* que devolve um array com a contagem dos *lengths.* O objetivo é determinar o número de vezes que um tamanho se repete.

Criamos uma função *generate_codes* que devolve um array (em formato de *hashmap*) com os respetivos códigos.

Como se processa a geração de códigos?

Inicialmente temos um número a zero, que corresponde ao valor do código a converter. Para um determinado comprimento, convertemos o número para binário, adicionamos-o ao array de códigos e somamos um ao mesmo. Quando geramos todos os códigos para esse tamanho, fazemos um *shift* de um bit para a esquerda e passamos ao próximo tamanho.

Por fim, criamos uma árvore de *Huffman* e adicionamos os códigos gerados.

Exercício 4 e 5

Para a leitura dos comprimentos e das distâncias criamos a função *read_huffman_trees.*

Como se processa a leitura dos comprimentos dos *lengths* e *distances*?

Primeiramente, construímos um código que esteja na árvore criada anteriormente. Para isso, lemos bit a bit e verificamos se o código está presente na árvore.

Quando obtido o código temos quatro hipóteses:

- Código inferior a 16:
  - Adicionamos o código na posição respetiva.
- Código igual a 17:
  - Lemos três bits adicionais;
  - Somamos três ao valor lido;
  - repetimos o código anterior esse número de vezes.
- Código igual a 17:
  - Lemos três bits adicionais;
  - Somamos três ao valor lido;
  - repetimos o código zero esse número de vezes.
- Código igual a 18:
  - Lemos sete bits adicionais;
  - Somamos onze ao valor lido;
  - repetimos o código zero esse número de vezes.

No final temos um array (no formato de *hashmap*) com o comprimento dos códigos.

Por fim, criamos uma árvore de *Huffman* e adicionamos os códigos gerados.

NOTA: Esta função está genérica o suficiente para ser usada para *HLIT's* e *HDIST's.*

Exercício 6

Fizemos uso do trabalho desenvolvido no exercício 3.

Exercício 7

Inicialmente, usamos a função *search_node* para obtermos um código onde temos três decisões a tomar:

- Código menor do que 256:
    - Representa o caractere ascii;
    - Adicionamos ao array do texto.
- Código igual a 256:
    - Chegamos ao final da codificação.
- Código maior do que 256:
    - Temos o array "*bits"* que diz-nos quantos bits temos de ler e o array "*lengths"* correspondentes ao comprimento;
    - Obtemos as distâncias da mesma forma;
    - Adicionamos o *sliced array,* obtido com o comprimento e distância ao texto.

Exercício 8

No final do exercício 7, obtivemos um array com o texto. No entanto, este array continha o valor *ascii* de cada caractere e não o caractere.

Sendo assim, usamos a função chr para converter os valores para caracteres e escrevemos num ficheiro: "output.txt".

Conclusão

O facto de o deflate ser uma algoritmo patenteado e explicações sobre o algoritmo são difíceis de encontrar pela internet, com ajuda da professora das aulas laboratoriais, conseguimos aprofundar diversos conceitos estudados nas aulas teóricas, entre os quais:

- Codificação usando árvores de Huffman
- Descompactação de dados comprimidos