Teoria da Informação
2º Trabalho Prático
CODEC não destrutivo para Texto

Eduardo Nunes, *2020217675,* André Moreira, *2020239416* e Diogo Tavares, *2020236566*

Abstract—Neste trabalho prático exploramos conceitos de Teoria de Informação, em particular, relativos à Teoria da Compressão. Pretendemos responder, principalmente, à questão: Quais serão os algoritmos mais eficazes de compressão não destrutiva de texto e as suas principais diferenças?
Para isto, calculamos o limite mínimo teórico para o número médio de bits por símbolo, a Entropia, para cada uma das fontes fornecidas (todas baseadas em documentos de texto) e, seguidamente, comprimimos essas fontes com diversos algoritmos de compressão de texto monitorizando, por fim, o número médio de bits por símbolo, verificando assim, quais destes se aproximam mais da Entropia de acordo com o CODEC utilizado. Os algoritmos ideais foram aqueles que se aproximaram mais da Entropia consistentemente nas diversas fontes e não abdicaram de muito poder e tempo de processamento para a sua codificação.

Index Terms—Teoria da Informação, IEEE, CODECs, LaTeX, algoritmos, compressão de texto, Python.

1 Introdução

1.1 Problema e a sua importância

Este trabalho visa encontrar uma solução para a compressão eficiente não destrutiva de texto. Um dos pilares da vida humana é **transmitir informação o mais rapidamente possível**, primeiro fisicamente através de cartas, sinais luminosos, sinais sonoros, impulsos elétricos e na **atualidade essencialmente através de bits digitais**. A determinada altura **é certo de que iremos chegar a um limite físico da rapidez com que enviamos bits**. Se continuarmos a necessitar de enviar grandes quantidades de informação com cada vez maior rapidez, é necessário encontrar uma maneira de representar a mesma informação com menor número de bits – **é crucial comprimir a data a enviar.**

1.2 Implementação

É de realçar que **o foco desta pesquisa não reside na implementação propriamente dita, mas sim na sua fundamentação e validação**. Nesse sentido, decidimos demonstrar o código dos diferentes CODECs[1] com base na linguagem *Python* devido à sua facilidade de leitura, à sua fácil implementação, caso o leitor pretenda testar os algoritmos apresentados, e ao fácil acesso de módulos que podem auxiliar o tratamento de dados das fontes.

1.2.1 Dados por base

Devido a esta notável questão, pretendemos, usando 4 fontes de texto distintas como exemplo:

1) *bible.txt*: Ficheiro de texto composto por uma transcrição da Bíblia. A sua entropia é 4.34275 bits/símbolo.
2) *finance.csv*: Ficheiro de texto composto por data separada por vírgulas. A sua entropia é 5.15995 bits/símbolo.
3) *jquery-3.6.0.js*: Ficheiro de texto composto pelo código fonte da livraria de *JavaScript*:*Jquery*. A sua entropia é 5.06698 bits/símbolo.
4) *random.txt*: Ficheiro de texto composto por um conjunto de caracteres aleatório. A sua entropia é 6 bits/símbolo.

Para codificar estas fontes é indispensável referir que haverá CODECs melhores do que outros de-

- *Docente da cadeira: Paulo Fernando Pereira de Carvalho.*
- *Docente da aula: Marco António Machado Simões*
  Grupo 5 da turma PL6.

Trabalho realizado com base nos conhecimentos obtidos na cadeira da Teoria da Informação.

[1]CODEC significa "Coder-Decoder", algoritmo de compressão de data de modo a transmiti-la mais rapidamente.

pendendo da fonte selecionada. Pretende-se, portanto, encontrar não um CODEC ideal para uma determinada fonte, mas sim **o mais eficaz geralmente**.

1.3 *State of the art*

Como foi referido inicialmente, para o ser humano viver em sociedade é indispensável transmitir informação o mais rapidamente possível, por isso, é essencial haver um constante avanço (nem que seja mínimo) por parte dos estudos da matéria em causa. O uso de um algoritmo ideal de compressão sem perdas de data é, obviamente, muito dependente do tipo de data em causa. Atualmente, é essencial ter em conta o tipo de data a codificar na sua codificação, apesar de existirem CODECs não destrutivos para todo o tipo de data em geral.

1.3.1 Codificação ***Delta Encoding***

Um bom exemplo da importância da dependência de data é visto quando se codifica uma imagem. Como se trata de uma imagem o método *PNG*[2] usa uma versão de *Delta Encoding* ao fazer *Filtering*, onde em vez de representar todos os valores de uma dada fonte, representamos a diferença entre eles o que é muito útil em imagens visto que os pixeis vizinhos estão usualmente correlacionados.

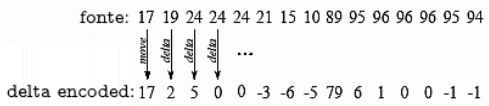

Fig. 1. Exemplo de uma codificação atual para imagens: *Delta Encoding*, a data codificada é obtida pela a diferença do valor anterior. Em imagens (no formato *PNG*), em vez de números á direita, tem-se pixeis com diferentes valores de cores/posições.

1.3.2 Codificação de ***Huffman***

O Código de Huffman, um código ótimo devido a nenhum código ser prefixo[3] um do outro, é sem dúvida um dos métodos de compressão mais conhecidos devido à sua simplicidade, fácil implementação, eficácia e não ter patente daí esta codificação ser amplamente usada em aplicações de compressão que vão de *GZIP*, *PKZIP*, *BZIP2* a formatos de imagem como *PNG* e *JPEG*.

Esta codificação consiste em:

1) Ordenar os símbolos por ordem crescente de ocorrências
2) Construir uma árvore binária até não haver mais símbolos:
   a) Combinar os dois símbolos menos frequentes num único símbolo (cada folha é um símbolo sendo, portanto, um bit com um valor de 1 ou 0)
   b) Acrescentar o novo símbolo à lista em que a frequência é a soma das frequências individuais.

A eficácia desta codificação reside no facto dos símbolos que ocorrem mais vezes sejam codificados com menos bits, estando os símbolos com mais ocorrências mais próximos da raiz da árvore binária.

A eficiência do código é dada por:

$$H(S) \leq \bar{l} < H(S) + 1$$

Sendo $H(S)$ a entropia, $\bar{l}$ o número de bits médio.

Apesar da sua grande eficiência, infelizmente, é de grande importância referir que não é perfeito. Além de também termos que codificar a árvore binária gerada depois de codificar a fonte o código de Huffman assume que:

1) A data usada é independente, ou seja, que os valores a comprimir são independentes, o que geralmente não são[4].
2) Tem de existir o modelo da distribuição estatística. Uma resposta para este problema foi o *Modelo Adaptativo de Huffman*.

Para terminar, uma tentativa de optimização da **codificação de Huffman** foi tentar agrupar vários símbolos de modo a aumentar a dependência interna entre eles, no entanto, não se costuma fazer isto porque se um alfabeto tiver m símbolos, um agrupamento n implica um alfabeto com m^n símbolos (o número de símbolos cresce exponencialmente o que em termos de memória não é ideal). Uma solução usada foram os *Códigos Aritméticos*.

1.3.3 Codificação ***LZW***

A codificação LZW[5], uma variante do LZ78[6] em que se evita o envio duplo do "codificador" visto que, com este método, o codificador é construindo progressivamente ao contrário da variante *LZ78*.

[2]significa *"Portable Network Graphics"*

[3]Um código diz-se "de prefixo" quando nenhuma palavra é prefixo da outra, ou seja, um código de prefixo é um código unicamente descodificável.

[4]Como foi referido no inicio do deste Capitulo a dependência da data é muito importante quando queremos reduzir a entropia, é importante saber o tipo de data que queremos comprimir porque sabemos que dependências tomar partido de para obter uma codificação ideal

[5]Por extenso, *Lempel-Ziv-Welch*.

[6]Outro método de compressão não destrutivo, feito em 1978 por Lempel-Ziv

O método consiste em:

1) Criar um dicionário com todos os símbolos do alfabeto.
2) Quando surge um novo padrão constituído $a|b$ sendo que a já se encontra no dicionário e ab não, encontrar a maior sequências de símbolos a da fonte existentes no dicionário.
3) Transmite-se o índice do dicionário e cria-se uma entrada nova ab no dicionário, ficando assim disponível a sequência ab para a sua próxima ocorrência.
4) Finalmente, continua-se a codificação no símbolo b até acabar a cadeia a codificar.

É importante referir que este CODEC foi **um dos primeiros métodos de codificação de texto não destrutivos universalmente usado por computadores**. Um documento de texto em inglês de grande dimensão, ao aplicarmos esta codificação, pode reduzir o seu tamanho original por metade! Apesar da sua implementação ser um bocado complicada por causa da gestão do dicionário e, em termos de eficiência, ao ser lido um novo caracter, ter que percorrer todo o dicionário à procura das formações deste com a cadeia pode ser desgastante para o CPU em causa...

1.3.4 Codificação ***Run-Lenght-Encoding***

Este método consiste em representar sequências de valores iguais seguidos de forma eficiente. O algoritmo, aplicável se o comprimento da sequência for maior que 3 para evitar aumentar a data inicial, consiste em substituir conjuntos de letras seguidas por o seu número de ocorrências em vez de elas todas, como por exemplo:

Data	aaabbbaaaaabbc
Comprimida	3a3b5a2b1c

1.3.5 Codificação ***Burrows-Wheeler***

A codificação **Burrows-Wheeler** consiste em, dada uma cadeia, adicionamos um caracter de controlo no final dela, criamos todas as rotações (sendo uma rotação a troca do ultimo carácter em primeiro lugar) da data, organizamos todas as colunas de rotações alfabeticamente e por fim retiramos a nossa cadeia codificada, sendo esta a ultima coluna das rotações ordenadas alfabeticamente. Este método, apesar de ser preciso guardar um caracter a mais (o caracter de controlo adicionado no final da cadeia em primeiro lugar) para ser feita a descompressão faz com que os caracteres repetidos fiquem muito próximos uns dos outros, **criando um ambiente propicio para o uso de outro método por cima deste, tal como o RLE**[7].

1.3.6 Codificação ***Move-to-Front***

Este método de compressão faz com que os caracteres repetidos mais vezes na cadeia fiquem no principio do alfabeto e a cadeia codificada.

Como conseguimos ver com o exemplo da codificação da palavra "*panama*":

Adicionado	Sequência	Lista do Alfabeto
3	p	amnp
1	pa	pamn
3	pan	apmn
1	pana	napm
3	panam	anpm
1	panama	manp

A lista codificada final fica: $[3, 1, 3, 1, 3, 1]$

1.4 Codificações a usar

Para o nosso trabalho usaremos as seguintes combinações:

1) **Codificação de Huffman**
2) **Codificação LZW + Huffman**, uma junção que normalmente resulta em compressões muito boas, visto que depois de codificar em LZW, um conjunto de bits dado da codificação LZW, como por exemplo, 000 pode ser representado por um só bit 0 se aplicarmos Huffman a seguir.
3) **Codificação Burrows-Wheeler + Run-Lenght-Encoding**, uma combinação muito eficaz também visto que ao aplicarmos o CODEC Burrows-Wheeler tendemos a agrupar os caracteres iguais, o que é ideal para, seguidamente, junta-los todos usando apenas um número com a codificação *RLE*.
4) **Codificação Move-to-Front + Huffman**, outra combinação excelente porque a codificação **Move-to-Front** faz com que os caracteres que estão a ser comprimidos se transformem numa sequência de números de "movidas", sendo uma "movida" os passos que um caracter faz da sua posição atual no alfabeto até ao seu principio. A lista gerada, com isto, é normalmente uma lista com elementos repetidos diversas vezes, o que torna ideal a utilização da codificação de Huffman que codifica idealmente essas repetições para estarem com o menor número de bits possíveis.

[7] Run-Lenght-Encoding.