Compressão não destrutiva

Trabalho pratico nº2

Eduardo Figueiredo  
2020213717

Departamento de Engenharia Informática

Faculdade de Ciências e Tecnologias da Universidade de Coimbra

Coimbra, Portugal

Fábio Santos  
2020212310

Departamento de Engenharia Informática

Faculdade de Ciências e Tecnologias da Universidade de Coimbra

Coimbra, Portugal

Filipe Soares  
2020238986

Departamento de Engenharia Informática

Faculdade de Ciências e Tecnologias da Universidade de Coimbra

Coimbra, Portugal

**Abstract –**Cada vez mais se fala em termos de eficiência e produtividade nos mais diversos campos.

A indústria da tecnologia não é exceção, conforme a informação que é necessário transmitir, sendo indispensável a utilização de um algoritmo de compressão eficaz e não destrutivo. Assim, analisámos diversos métodos e algoritmos de compressão sem que o conteúdo original fosse destruído.

**Palavras-chave: compressão, codificação, algoritmos, Lossless**

# Introdução

No intuito de criar uma compressão não destrutiva para texto, este documento irá abordar as várias técnicas de compressão de texto, abordando os prós e os contras do uso de cada uma. Para isso, analisar-se-ão as velocidades de compressão de cada uma delas, bem como a eficiência destas. Uma vez que os ficheiros fornecidos são ficheiros de texto, abordaremos apenas métodos “Lossless”, isto é, compressões que não percam qualquer tipo de informação.

O nosso foco irá centrar-se nas técnicas lecionadas nas aulas assim como nos ficheiros disponibilizados pelos docentes da cadeira.

# Estado da arte

Iremos abordar dois tipos de métodos de compressão de dados: os métodos estatísticos e os métodos aritméticos (baseados na entropia).

## Códigos de Huffman

Este método de compressão foi desenvolvido por David Albert Huffman (pioneiro na ciência da computação) em 1952. É um algoritmo de compressão eficiente em texto e fundamenta-se em:

1. Percorre-se pelos dados e regista-se a frequência com que cada símbolo ocorre numa tabela.
2. Com os dois símbolos de menor frequência, cria-se uma arvore de nó de duas folhas e junta-se o valor das frequências dos símbolos como se se tratasse de um nó individual, repetindo-se o mesmo até existir um, único nó.

Desta forma, os símbolos mais frequentes são representados por códigos mais curtos e os menos frequentes representados por códigos mais longos.

## Códigos Aritméticos

É um algoritmo baseado na entropia que se fundamenta em substituir um fluxo de símbolos de entrada com um único número de virgula flutuante como saída. O conceito básico da codificação aritmética foi desenvolvido por Elias em 1960 e posteriormente desenvolvido em grande parte por Pasco, Rissanen e Langdon.

O principal objetivo da codificação aritmética é atribuir um intervalo para cada símbolo potencial. Então um número decimal é atribuído a este intervalo. O algoritmo começa com um intervalo de 0,0 a 1,0. Após cada símbolo de entrada do alfabeto é lido, o intervalo é subdividido em um menor intervalo em proporção à probabilidade do símbolo de entrada.

Este subintervalo então se torna o novo intervalo e é dividido em partes de acordo com a probabilidade de símbolos do alfabeto de entrada. Isso é repetido para cada símbolo de entrada. E, no final, qualquer número do intervalo final determina exclusivamente os dados de entrada.

## LZ-77

Jacob Ziv e Abraham Lempel apresentaram o seu método baseado em dicionário em 1977 que funcionava através da exploração de palavras, frases ou carateres repetidos. É definido um buffer como janela flutuante com o objetivo de contar as ocorrências de cada símbolo.

Quando é detetada uma repetição de um ou mais símbolos é guardado um ponteiro de 2 bytes que aponta para a última ocorrência desse símbolo /conjunto de símbolos em que a referência será algo do tipo <i,j> onde i é o deslocamento para a correspondência da primeira ocorrência e j o comprimento da correspondência. Se não há correspondência, o algoritmo gera um ponteiro nulo. De notar que i e j são números de 8 bits e por isso não poderão ser maiores que 255.

## LZW (Lempel–Ziv–Welch)

Este algoritmo foi desenvolvido por Terry Welch, que resultou de um melhoramento do LZ77 e do LZ78 em 1984.

O algoritmo LZW é uma técnica de compressão que consiste em primeiro, criar um dicionário inicial usando códigos de 8 ou 12 bits, habitualmente usando código ASCII ou UTF8 cujas 256 linhas são os carateres. Seguido por ler símbolo a símbolo comparando sempre cada símbolo do ficheiro com os existentes no dicionário. Um exemplo disto é:

“LAILAA”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| S | c | Saída | Código | Sequência |
| - | - | - | 1 | A |
| - | - | - | 2 | I |
| - | - | - | 3 | L |
| L | A | 3 | 4 | LA |
| A | I | 1 | 5 | AI |
| I | L | 2 | 6 | IL |
| LA | A | 4 | 7 | LAA |

**Tabela 1 - Exemplo da codificação LZW**

Código comprimido = 3124 (números da coluna saída)

## Deflate

O Deflate trata-se de um algoritmo moderno que recorre a algoritmos como o LZ77 e Huffman Encoding. Este algoritmo desenvolve-se do seguinte modo:

Uma Stream Deflate consiste numa série de blocos. Cada bloco é precedido por um header de 3 bits.

1º bit (Marcador do último bloco): 0-Último bloco; 1-Não restam mais blocos a alcançar; 2º e 3º bits (Métodos de compressão): 00-Secção de armazenamento; 01-Árvore de Huffman pré acordada; 10-Bloco comprimido com árvore de Huffman; 11-Reservado (Não se usa); Ao aplicar LZ77 nestes blocos faz com que as Strings duplicadas sejam removidas. Finalmente, é gerada uma árvore de Huffman com espaço para 288 simbolos, porém, o 286 e o 287 não são utilizados.

## PPM(Prediction by Partial Matching)

Desenvolvido por Cleary e Witten em 1984, o PPM, é um algoritmo de compressão que demonstrou resultados muito bons no que toca a rácios de compressão apesar de não ser dos mais rápidos comparativamente a outros já existentes, pelo que para ficheiros grandes não é o mais indicado.

Este algoritmo começa por encontrar ocorrências de strings de um tamanho definido. Aplica o mesmo processo ao que sobrar, mas desta vez o tamanho da string é uma unidade inferior, isto é repetido até que todas as combinações sejam encontradas e associadas a uma probabilidade que esteja de acordo com a sua ocorrência. Por último são enviadas as probabilidades obtidas para o codificador aritmético adaptativo.

Em suma, o método apresenta uma boa viabilidade na compressão dos dados.

## Tranformada de Burrows-Wheeler

Este método de compressão de Burrows-Wheeler (BWT), foi inventado por Michael Burrows e David Wheeler em 1924. É usado em técnicas de compressão como por exemplo o Bzip e é um dos melhores algoritmos de compressão global para texto. A Transformada de Burrows-Wheeler funciona com blocos de dados e não em “Streams”, que leem uma pouca quantidade de dados de cada vez, adversamente aos algoritmos da família LZ. Quanto maior for o tamanho dos blocos, maior será a taxa de compressão obtida. Recebe como parâmetros de entrada um vetor A de tamanho N com os dados a serem transformados e retorna um vetor B e um índice k. A princípio, são geradas todas as N rotações à direita para o vetor A, dando origem a uma matriz N x N. De seguida, as N linhas são ordenadas e são copiados o último caracter de cada uma para o vetor B. A k é concedido o número da linha da matriz ordenada correspondente à A.

## MTF(move to front)

Este método de compressão foi desenvolvido por Bentley, Sleator, Tarjan e Wei em 1986. É um método que reorganiza a informação para ser mais compressível, sendo usada como um passo intermedio em alguns algoritmos de compressão.

Em primeiro lugar, este algoritmo tem um alfabeto de todos os símbolos a procurar. Posteriormente, os símbolos irão ser substituídos por símbolos na sua posição respetiva no array de output.

Por último, será realocada para o início a posição desse símbolo no alfabeto.

Assim, os símbolos mais frequentes estarão no início do alfabeto.

## Bzip2

Desenvolvido por Julian Seward em 1996. Este método começa por subdividir a informação em blocos entre 100KB e 900KB, seguido por comprimir estes através da técnica de Burrow Wheeler. Também é possível usar métodos como o RLE e o Huffman.

Concluindo, este método é muito eficaz em termos de espaço e de tempo de compressão, dai ser bastante usado atualmente.

## RLE(Run\_Length\_Encoding)

Este método consiste em reduzir o tamanho criando pares com o símbolo e o número de repetições do símbolo. Um exemplo deste é:

“aaabdsssss” = 3ª1b1d5s.

Assim é um bom algoritmo para ficheiros que contenham várias repetições de símbolos.

# Vantagens e desvantegens

Uma vantagem dos Huffman codes é que não necessita de conhecer a estatística dos símbolos fonte e é uma codificação relativamente rápida e de acesso aleatório, pois a sua árvore é construída durante a compressão/descompressão. Porém, a principal desvantagem deste método é que esta árvore não é única.

A principal vantagem da codificação aritmética é facilidade de se implementar sistemas com múltiplos codificadores aritméticos e uma das desvantagens é que funciona com probabilidade comutativa.

O algoritmo LZ77 é um método muito eficiente para codificação de textos longos, no qual contenha várias palavras repetidas, porém a principal desvantagem é que quanto maior for o buffer maior é tempo de processamento e maior número de bits gastos no ‘i’ e no ‘j’.

A principal desvantagem do Deflate é se tentarmos comprimir um ficheiro com poucas repetições, o ficheiro comprimido irá ocupar um número semelhante de bytes do original.

O método de compressão PPM permite atingir bons rácios de compressão devido ao seu modelo estatístico, assim como, ao codificador aritmético, de que faz uso, porém não se trata de um codificador tão rápido a comprimir, tornando-se este a característica menos positiva deste método de compressão.

# Descrição do código

O código é constituído por um ficheiro main.py e vários ficheiros auxiliares, cada um deles contendo as funções de um método de compressão ou de codificação. De notar que os ficheiros auxiliares contêm código de outros autores adaptado por nós. Ao correr o ficheiro main, o utilizador depara-se com uma interface na consola onde pode escolher o método de compressão que pretende utilizar. O programa chama então as funções de compressão e de descompressão do ficheiro do método correspondente e no final é mostrado o tempo de compressão de cada ficheiro bem como o tamanho do ficheiro final e o rácio de compressão. Os ficheiros originais são lidos da pasta “./dataset” enquanto os ficheiros comprimidos são guardados na pasta”./resultados” e as descodificações na pasta “./decompress”.

# Resultados das compressões

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Métodos de compressão | bible.txt | finance.csv | jquery-3.6.0.js | random.txt |
| ORIGINAL (kbytes) | 3953 | 5744 | 282 | 98 |
| BZIP2 | 826 | 185 | 68 | 74 |
| Huffman | 2167 | 3703 | 180 | 74 |
| LZW | 2284 | 1315 | 172 | 144 |
| PPM | 730 | 177 | 61 | 76 |
| MTF | 1540 | 11428 | 565 | 196 |
| RLE | 7816 | 11103 | 521 | 193 |
| BWT | 3983 | 5744 | 293 | 98 |
| BWT + RLE | 2686 | 537 | 203 | 193 |
| LZW + BZIP2 | 1375 | 721 | 116 | 85 |
| BWT + PPM | 834 | 138 | 75 | 76 |
| Huffman + BZIP | 1140 | 416 | 111 | 74 |
| BWT + LZW | 1959 | 404 | 156 | 144 |

**Tabela 2 – Tamanho dos ficheiros após compressão**

Inicialmente, é de notar que, de todos os métodos utilizados, o BZIP2 e o PPM são os mais eficazes na compressão de texto uma vez que para todos os ficheiros, estes métodos são os que obtêm um rácio de compressão maior.

Porém, é de realçar, que cada ficheiro é um ficheiro e o(s) método(s) eficaz(es) para uns pode(m) não ser eficaz(es) para outro(s).

Métodos que tiram partido das repetições de símbolos, como o Huffman, acabam por ser eficazes em quase todos os ficheiros, uma vez que tanto a “bíblia.txt”, o “finance.csv” e o “random.txt” acabam por ter muitas repetições, sejam elas letras ou palavras.

O LZW é dos métodos mais demorados do conjunto de métodos testados, uma vez que tem de fazer um dicionário de carateres e verificar se o próximo carater está nele. Tirando esta desvantagem, nos ficheiros “random.txt” e “finance.csv” acaba por atingir um bom rácio de compressão (mais de metade em ambos os casos) uma vez que há conjuntos de letras iguais seguidos nestes ficheiros, o que não acontece nos outros ficheiros, o que acaba tendo um impacto negativo no ficheiro “random.txt” visto que, após a “compressão”, o ficheiro resultante é mais pesado que o original.

O RLE, quando aplicado sem nenhum outro método, é o método menos eficaz dos demais uma vez que ele tira partido de repetições do mesmo símbolo seguido, o que é raro em ficheiros de texto. Assim sendo, quando aplicado sozinho, os ficheiros obtidos acabam por ficar maiores do que os ficheiros originais.

O BWT é um método de codificação de texto, que, quando aplicados métodos de compressão, acaba por ter um melhor resultado do que o método de compressão sozinho. É o caso do BWT seguido do RLE, do BWT seguido do PPM e o BWT seguido do LZW. Todos estes conjuntos obtêm um melhor resultado do que o método de compressão atuando

sozinho.

# Análise de resultados

|  |  |  |
| --- | --- | --- |
| Ficheiro | Melhor método(s) de compressão | Rácio de compressão |
| bible.txt | PPM | 5.42 |
| finance.csv | BWT + PPM | 41.62 |
| jquery-3.6.0.js | PPM | 4.55 |
| random.txt | BZIP2 / Huffman | 1.32 |

**Tabela 3 – Melhor resultado para cada ficheiro**

##### Referências

1. <http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman/>
2. <http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/codificacao-aritmetica/>
3. <http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lz77/>
4. A\_Comparative\_Study\_Of\_Text\_Compression\_Algorithms
5. Comp com20200113-81589-6qysrz -v2
6. Modeling for Text Compress
7. <https://github.com/chaitan94/knit>
8. <https://github.com/bhrigu123/huffman-coding/blob/master/useHuffman.py>
9. https://github.com/amycardoso/lzw-text-file-compression/blob/master/lzw.py
10. <https://titanwolf.org/Network/Articles/Article?AID=2fb527a4-b1a2-4e05-8f56-ad861e0f480e>
11. https://ppmd-cffi.readthedocs.io/\_/downloads/en/stable/pdf/
12. https://stackabuse.com/run-length-encoding/