Codecs lossless de texto

Pedro Martins

Universidade de Coimbra

Coimbra, Portugal

Pedro.afonso2001.pm@gmail.com

João Silva

Universidade de Coimbra

Coimbra, Portugal

j0a0carll21@gmail.com

***Abstract*— Estado de arte sobre os vários tipos de codecs lossless para a compressão de texto e respetivas vantagens e desvantagens associadas a cada um dos algoritmos mencionados.**

Keywords*— *lossless, codecs, RLE, Huffman Code, LZ_77, LZ_78, LZW, GOLOMB, Deflate.

I. Introdução

Ao longo deste estado de arte iremos analisar os diversos métodos de codificação de texto, de forma a reduzir drasticamente o tamanho ocupado pelos mesmos. Estes métodos serão do tipo *lossless* uma vez que não irão alterar de forma significativa a qualidade do texto. Também iremos proceder à análise de diversos algoritmos concebidos com o intuito de ajudar no processo de compressão.

II. CODECS LOSSLESS ANTIGOS

Existem diversos codecs que procedem à codificação de informação sem perdas de qualidade. Nesta lista estão representados os algoritmos mais antigos.

A. *Run Lenght Encoding (RLE):*

Consiste na compressão de um texto de forma sequencial. Procura repetição de símbolos de forma a diminuir a repetição do mesmo sinal e assim levar a um aumento da compressão e respetiva diminuição do tamanho**[1]**. Na Figura 1 está presente um exemplo da aplicação deste algoritmo.

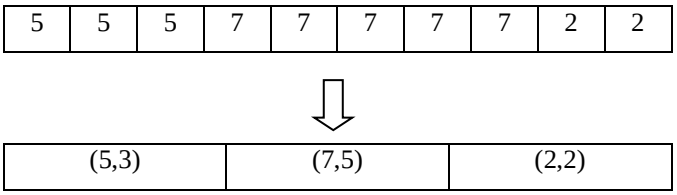

Figura 1 – Exemplo do RLE

Este tipo de compressão não é o mais eficaz contudo, é relativamente simples de executar e leva a uma diminuição do ficheiro final. Os resultados dependem bastante de ficheiro para ficheiro.

B. *Huffman Code:*

É um método de compressão que contabiliza um número de ocorrências de cada simbolo e cria uma árvore binária a partir desses dados criando códigos de tamanho variável para cada símbolo**[2,3]**.

Primeiramente, os símbolos são ordenados por ordem crescente. De seguida, procede-se à construção de uma árvore binária onde os dois símbolos menos frequentes são combinados num único símbolo (folhas). Por fim, acrescentar o novo símbolo à lista em que a frequência de ocorrência é a soma das frequências individuais. Esta iteração ocorre até que exista um único símbolo. Na Figura 2 está representado este algoritmo.

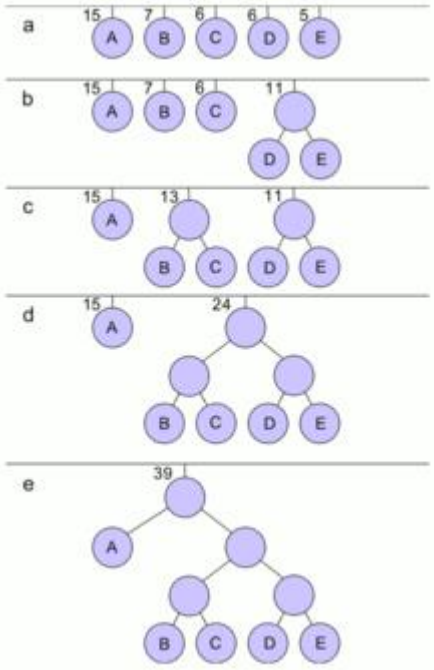

Figura 2 – Representação de uma árvore de Huffman**[2]**

Algumas vantagens a apontar**[10,11]**:

- Não necessita de conhecer a estatística dos símbolos fonte.
- Se a estatística variar, o código adapta-se automaticamente.

Algumas desvantagens a apontar:

- O poder obter diversas árvores binárias para os mesmos dados de entrada
- Pouca tolerância a falhas de descodificação
- Pouco eficiente para alfabetos reduzidos

C. LZ_77:

O algoritmo de LZ77**[4]** baseia-se na utilização das partes que já foram lidas de um arquivo como um dicionário, substituindo as próximas ocorrências das mesmas sequências de símbolos pela posição (absoluta ou relativa) da sua última ocorrência. Sendo ainda mais eficaz para texto do que para imagens

Segue-se o algoritmo de funcionamento para a codificação LZ_77:

NOTAS PARA A LEITURA DO ALGORITMO:

- A janela lê-se da esquerda para a direita mas o indice aumenta da direita para a esquerda.

- O buffer lê-se da esquerda para a direita e o indice aumenta da esquerda para a direita.

- A sequência tem de começar desde o primeiro símbolo do buffer e é sempre a maior existente.

- Indice da sequência é onde a sequência começa pela primeira vez.

Enquanto não chegar ao fim do ficheiro:
dicionário = null
buffer = n primeiros símbolos (tamanho n bytes)
janela = null (tamanho m bytes)

Enquanto existir símbolo para codificar:
Se existir uma sequência presente no buffer que esteja também na janela:

c = próximo símbolo depois da sequência
Enviar código (i(sequência), len(sequencia, c)
janela = janela + sequência + c
atualizar dicionário com sequencia + c

Senão:

c = primeiro símbolo no buffer
Enviar código (0,0 c)
janela = janela + c
atualizar dicionário com c

buffer = n primeiros símbolos depois de c

De seguida está representado um exemplo do método de funcionamento deste algoritmo:

Janela	Buffer	Resto	Tuplo	Detetado	Somado
	A_AS	A_DA_CASA	(0,0,A)	---	A
A	_ASA	_DA_CASA	(0,0,_)	---	_
A_	ASA_	DA_CASA	(1,1,S)	A	AS
A_AS	A_DA	_CASA	(3,2,D)	A_	A_D
A_ASA_D	A_CA	SA	(2,2,C)	A_	A_C
ASA_DA_C	ASA		(7,3,EOF)	ASA	ASA

Figura 3 – Exemplo do codec LZ77

Algumas desvantagens a apontar: a dimensão do lookahead buffer condiciona a máxima dimensão da sequência a codificar, quanto maior a janela maior a compressão mais tempo de processamento e aumento do número de bits gastos nos componentes posição e tamanho.

D. LZ_78:

O algoritmo de LZ78**[5,6]**, à semelhança do LZ77, usa um dicionário para armazenar as sequências de símbolos encontradas no arquivo, e usa os códigos (posição das sequências no dicionário, ou mesmo um número atribuído sequencialmente à sequências de caracteres encontradas). Sendo ainda mais eficaz para texto do que para imagens.

Segue-se o algoritmo de funcionamento para a codificação LZ_78:

Enquanto não chegar ao fim do ficheiro:
dicionário = null
word = null
c = null

Enquanto existir símbolo para codificar:

Se word + c é uma sequência presente no dicionário:
word = word + c
c = próximo símbolo

Senão:

Enviar código (índice(word), c)
Atualizar dicionário com word + c
word = null

E. LZW:

A codificação LZW**[7]** baseia-se num dicionário que contém os diversos símbolos a ser codificados. No início, o dicionário encontra-se vazio, começando a aumentar de tamanho à medida que novas combinações de símbolos vão sendo obtidas ao ler o texto. Quando se encontra uma combinação de símbolos com o mesmo tamanho e a mesma ordem de símbolos, irá-se adicionar 1 nova ocorrência à sua entrada no dicionário. Assim, no final da leitura, o ficheiro final encontra-se mais comprimido e ocupa menos espaço.

Este método de codificação é ideal quando existem diversos padrões de símbolos que se repetem no texto e que se encontram vastamente distribuídos. Também é uma técnica relativamente fácil de ser aplicada e que apresenta uma rápida taxa de compressão. Contudo é uma das técnicas sem perda de qualidade mais antigas e é propícia a criar entradas de dicionário que nunca são usadas, diminuindo a compressão total do ficheiro. Esta técnica é mais usada na compressão de texto face a outros tipos de ficheiros.

F. GOLOMB:

Este algoritmo[8] é um dos algoritmos mais simples existentes. Precisando apenas de um parametro inicial M, escolhido por nós, consoante o tipo de compressão que procuramos, é calculada uma parte q e uma parte r do número inicial as quais no final serão concatenadas num novo código que de certo ocupará menos bits que o original.

Segue-se o algoritmo de funcionamento para a codificação GOLOMB:

```
Enquanto não chegar ao fim do ficheiro:
  array_N = null
  M = m (escolher um número natural)
  b = floor(log2 (M))

  Enquanto existir símbolo para codificar:
    q = floor(c/M)*1+0
    r = n mod M
    Se r < 2^b-M:
      r usa b bits
    Senão:
      r = r + 2^(b+1) - M usando b bits (soma)
    Atualizar array_N com q + r (concatenação)
```

De seguida está representado um exemplo da codificação GOLOMB:

Se quisermos codificar por exemplo o número n=16:

q = floor(n/M) = floor(16/10) = 10
r = n mod M = 16 mod 10 = 6 => 6+2^4-10 = 12
valor final = q + r = 10 + 1100 = 101100

III. CODECS LOSSLESS MAIS RECENTES:

Segue-se uma técnica utilizados na compressão de texto sem perda de qualidade.:

G. Deflate:

Método de compressão[9] que utiliza LZ77 e Huffman Code. A partir deste algoritmo conseguimos bons racios de compressão. A maior desvantagem é que se tivermos um ficheiro com poucas repetições este algoritmos será pouco eficaz, pois irá se gastar quase tantos bits como o original.

É um algoritmo rápido em termos de compressão e decompressão mas traz um baixo rácio de compressão.

EXPLORAÇÃO DE MÉTODOS

Vamos analisar vários métodos baseados em algoritmos já referidos previamente. Estes métodos foram concebidos na linguagem de programação Python e vão ser testados num dataset de 4 ficheiros de texto com múltiplos formatos com o objetivo de descobrir qual a taxa de compressão e rapidez de compressão/descompressão associados aos respetivos métodos. Como tal a Figura 4 representa a comparação dos 4 ficheiros originais antes e após lhes ser aplicados os diversos métodos de compressão lossless:

	Bible.txt	Finance.csv	Jquery.js	Random.txt
Original	3953	5744	282	98
RLE	7768	11139	513	193
LZ77	3150	1507	222	140
LZW	2282	1318	171	142
Huffman	2167	3743	180	74
Deflate	2458	1196	174	105
RLE + Huffman	4316	7456	354	147

Figura 4 – Tamanho final do ficheiro comprimido usando vários métodos (em KB)

TAXA DE COMPRESSÃO

A taxa de compressão consiste numa valor percentual que representa o quão comprimido um ficheiro ficou em comparação com o seu tamanho inicial. Uma das grandes utilidades desta taxa é a possibilidade de ajudar a aferir o quão eficaz um método é a comprimir um ficheiro de texto e, a partir desse valor, estabelecer um ponto de comparação entre os vários métodos a ser analisados.

Para este cálculo iremos recorrer à seguinte fórmula (TamanhoComprimido – TC e TamanhoOriginal - TO):

TaxaDeCompressão(%) = 100 – (TC / TO) X 100

Na Figura 5 estão presentes os diversos valores para a taxa de compressão usando os valores dos tamanhos finais de cada ficheiro após a sua compressão que se encontram presentes na Figura 4 para proceder a esse cálculo:

	Bible.txt	Finance.csv	Jquery.js	Random.txt
RLE	-96.96	-93.92	-81.91	-96.24
LZ77	20.31	73.76	21.28	-42.86
LZW	42.27	77.05	39.36	-44.89
Huffman	45.22	34.84	36.17	24.49
Deflate	37.82	79.18	38.30	-7.14
RLE + Huffman	-9.18	-29.81	-25.53	-50.00

Figura 5 – Comparação das taxas de compressão (%)

Ao analisar os resultados da Figura 5 concluimos que o método Deflate, é o que apresenta uma melhor taxa de compressão, com um valor médio (obtido através da média da taxa de compressão dos 4 ficheiros de texto presentes no dataset original) de 37.04%, superando os 28.45% do LZW, os -92.43% do RLE (Run Length Encoding), os 18.12% do LZ77, os 35.18% do Huffman e, finalmente, os -28.63% da combinação dos algoritmos RLE e Huffman Encoding.

Um dos principais motivos por detrás do destaque do Deflate é o facto de este método de compressão recorrer a uma combinação de dois métodos, o LZ77 e o Huffman Encoding, o que resulta num aumento da taxa de compressão.

Também é possível constatar que o RLE em todos os ficheiros levou a um aumento do tamanho do ficheiro final, o que se pode justificar através do facto de que este método é muito mais bem sucedido na compressão de sequências consecutivas de texto face à compressão de texto corrido.

O ficheiro que, no geral, apresentou melhores resultados relativamente à sua compressão final foi o finance.csv, o que pode ser justificado através do tipo de ficheiro de texto que o mesmo representa (.csv usado em Excel), uma vez que, em comparação com, por exemplo, os dois ficheiros de texto .txt fornecidos no dataset, as taxas de compressão apresentam valores bastante inferiores.

RAPIDEZ DE COMPRESSÃO

A rapidez de compressão consiste no tempo que o ficheiro demora a ser comprimido. Este valor varia de método para método pelo que, quanto menor for a rapidez de compressão, mais eficiente é a sua aplicação na compressão de ficheiros de texto sem perda de qualidade. A Figura 6 estabelece uma comparação entre a rapidez de compressão para os métodos a ser estudados:

	Bible.txt	Finance.csv	Jquery.js	Random.txt
RLE	2.31s	3.69s	0.16s	0.06s
LZ77	22.34s	8.26s	1.47s	0.91s
LZW	1.26s	1.80s	0.08s	0.03s
Huffman	1.03s	1.63s	0.07s	0.02s
Deflate	23.56s	8.97s	1.48s	1.03s
RLE + Huffman	2.47s	4.09s	0.18s	0.08s

Figura 6 – Comparação da rapidez de compressão

Pela Figura 6 podemos constatar que o método do Huffman é o algoritmo mais rápido na compressão de imagens.

Logo, na soma dos parâmetros já considerados o algoritmo Huffman é aquele que, até agora, apresenta os resultados mais satisfatórios

RAPIDEZ DE DESCOMPRESSÃO

A rapidez de descompressão consiste no tempo que o ficheiro demora a ser descomprimido. Este valor varia de método para método pelo que, quanto menor for a rapidez de descompressão, mais eficiente é a sua aplicação na descompressão de ficheiros de texto sem perda de qualidade. A Figura 7 estabelece uma comparação entre a rapidez de descompressão para os métodos a ser estudados:

	Bible.txt	Finance.csv	Jquery.js	Random.txt
RLE	1.03s	1.39s	0.07s	0.02
LZ77	0.82s	0.49s	0.06s	0.03s
LZW	0.39s	0.21s	0.03s	0.02s
Huffman	5.22s	8.73s	0.44s	0.16s
Deflate	6.38s	3.17s	0.43s	0.25s
RLE + Huffman	6.56s	10.40s	0.50s	0.21s

Figura 7 – Comparação da rapidez de descompressão

Estes resultados apontam para o algoritmo LZW como o mais eficaz em termos temporais aquando da descompressão de um ficheiro de texto, contudo é ainda de notar que este algoritmo apresenta resultados muito positivos tanto na compressão (2º melhor) como na descompressão (melhor), ao contrário do Huffman que, apesar de ser bastante rápido a comprimir, apresenta tempos bastantes maus na descompressão do ficheiro (2º pior em termos temporais).

Em suma, na junção da consideração dos dois tempos (compressão e descompressão), considera-se que o LZW seja a melhor opção, o que pode ser justificado pelo facto de que o mesmo é uma versão melhorada de algoritmos já provados (LZ77 e LZ78) e extremamente eficiente na gestão do seu dicionário de sequências registadas.

COMPARAÇÕES FINAIS

Como parte do estudo dos codecs lossless de texto, iremos apresentar vários gráficos relativos às médias dos parâmetros atrás analisados de forma a poder tirar conclusões relativas ao melhor algoritmo a aplicar a compressão de ficheiros de texto.

Na figura 8 abaixo apresentam-se os valores médios da taxa de compressão do dataset original.

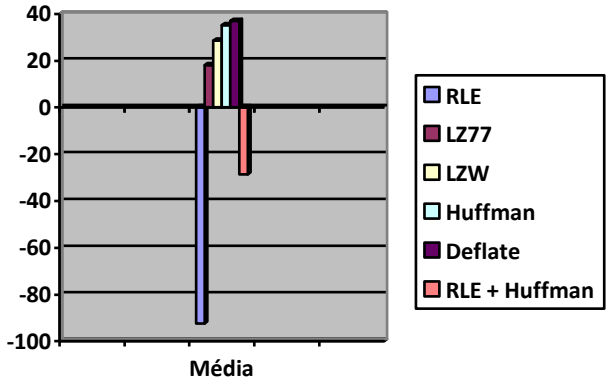

Figura 8 – Médias da taxa de compressão (Em %)

Na figura 9 abaixo apresentam-se os valores médios da rapidez de compressão do dataset original.

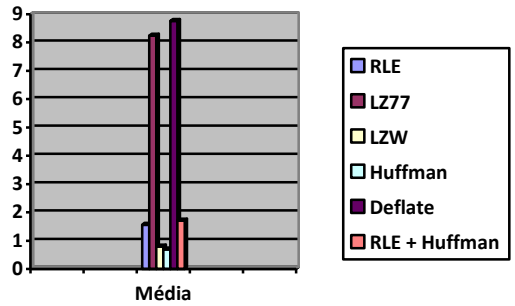

Figura 9 – Médias da rapidez de compressão (Em segundos)

Na figura 10 abaixo apresentam-se os valores médios da rapidez de descompressão do dataset original.

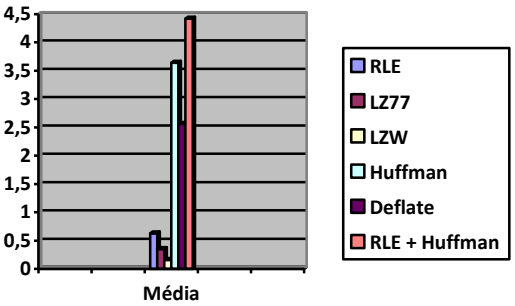

Figura 10 – Médias da rapidez de descompressão (Em segundos)

Com os resultados acima obtidos é possível isolar três codecs como os mais favoráveis a efetuar a compressão de texto, o LZW, Huffman Encoding e o Deflate. O Defalte é aquele que apresenta a melhor taxa de compressão. Já o Huffman Encoding é o que tem melhor rapidez de compressão contudo, a sua rapidez de descompressão é bastante má em comparação com o LZW. Assim, combinando estes fatores, o LZW é o melhor algoritmo no total dos parâmetros analisados, com o Huffman e o Deflate a apresentarem-se como sólidas escolhas.

Quanto ao pior algoritmo analisado, o RLE e a combinação de RLE + Huffman são as duas escolhas óbvias, com ambos a registar más taxas de compressão e o último a ter a pior rapidez de descompressão no dataset a ser analisado.

TRABALHO FUTURO

Os códigos desenvolvidos para os métodos de compressão LZ77, RLE, Deflate usam a função ‘bytes( )’ do Python o que nos limita à não utilização de números superiores que 256. Seria interessante arranjar uma melhor forma de armazenar as codificações feitas pelos nossos métodos de compressão.

Também se equaciona uma possível alteração à segunda etapa do processo, o Deflate. Será possível alterar o LZ77 por o mais recente LZ78 e se sim, quais seriam os ganhos associados a esta alteração.

CONCLUSÃO

Este estado de arte teve como foco explorar diversos tipos de codecs para uma compressão de texto sem perda de qualidade, explicando os seus algoritmos e módulos e apontando vantagens e desvantagens.

O estado de arte teve como foco explorar diversos tipos de codecs para uma compressão de texto sem perda de qualidade, explicando os seus algoritmos e módulos e apontando as respetivas vantagens e desvantagens.

De seguida criámos e analisámos diversos algoritmos baseados nos métodos apontados no estado de arte. Foi alvo de estudo a sua capacidade de compressão (taxa de compressão e rapidez de compressão e descompressão) num universo de 4 ficheiros de texto e os resultados explicados, comparados e justificados.

Por fim, estabeleceu-se uma comparação estes os mesmos algoritmos, com o intuito de descobrir qual atinge melhor compressão e porquê.

REFERÊNCIAS:

Abaixo seguem-se as referências para a construção do estado de arte.

[1] https://www.fileformat.info/mirror/egff/ch09_03.htm

[2] https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman

[3] Powerpoints das aulas Teoricas, Cap 2 - Teoria da Informação e Codificação (slide 101)

[4] https://pt.wikipedia.org/wiki/LZ77

[5] https://pt.wikipedia.org/wiki/LZ78

[6] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-baseada-em-dicionarios/lz78/

[7] https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch

[8] https://pt.wikipedia.org/wiki/C%C3%B3digos_de_Golomb

[9] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/codificador-qm/deflate/

[10] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman-adaptativo/

[11] http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman/

Material de pesquisa fornecido pelo professor.