Técnicas de compressão não destrutiva: CODEC não destrutivo para imagem e texto

Gabriel Fernandes, Maria Dias, Pedro Rodrigues

Departamento de Engenharia Informática, Universidade de Coimbra, Portugal

gabrielf@student.dei.uc.pt, mddias@student.dei.uc.pt, pedror@student.dei.uc.pt

***Abstract*- Devido ao crescente número de utilizadores de meios informáticos, a cada dia que passa é gerada mais informação que circula pela internet e/ou que é armazenada nos nossos dispositivos pessoais. Verificamos então que se torna importante encontrar formas de codificar a informação por forma a compactá-la facilitando a sua transmissão. Neste documento fizemos uma breve abordagem da eficiência de algoritmos de compressão sendo as nossas conclusões suportadas por estudos feitos sobre o Silesia Corpus. Concluímos que existem vários algoritmos mas que a nossa escolha deve ser feita tendo em conta o local da aplicação do algoritmo e as limitações do equipamento compressor. Tendo tudo isto em conta exploramos várias soluções de compressão não destrutiva num Dataset composto por um documento de texto (war_and_peace.txt) e numa imagem monocromática (cromenco_c10.bmp). Tentamos portanto diversas combinações de algoritmos de compressão tentando colocar preditores antes de alguns algoritmos explorados neste documento entre outros com o objetivo de tentar obter a maior compressão possível sendo os resultados apresentados e comentados neste documento.**

***Palavras chave*-** Algoritmos de compressão não destrutivos, Codificação, Silesia Corpus,Taxa de compressão, Velocidade de compressão/descompressão

I. INTRODUÇÃO

Os algoritmos de compressão de informação foram criados devido à necessidade de resolver problemas relacionados com a transferência de informação através da internet. A transmissão de informação nos canais existentes não atingia velocidades aceitáveis, devendo-se ao facto da largura de banda disponível não ser suficientemente grande para transmitir a informação em tempo útil. Assim, foi necessário diminuir o tamanho da informação, usando para o efeito técnicas de compressão.

Existem dois tipos de compressão: destrutiva e não destrutiva. A principal diferença entre ambos reside no facto de um permitir comprimir sem qualquer perda de informação (codificação não destrutiva), e outro leva a perda de informação no processo de compressão (compressão destrutiva). Enquanto que o primeiro método de compressão é baseado em técnicas que visam a reorganização da fonte de dados original através do uso de várias ferramentas matemáticas e estatísticas como a codificação entrópica, o segundo consegue obter uma compressão dos dados removendo bits que são desnecessários, reduzindo consequentemente o tamanho do ficheiro. Podemos assim concluir que após a descompressão não é possível obter o ficheiro inicial utilizando este método. A compressão de ficheiros é possível devido à redundância e dependência estatística dos símbolos do ficheiro fonte.

Neste documento vamos analisar as principais etapas de compressão, os algoritmos mais usados e as combinações mais usuais destes.

II. ETAPAS DA CODIFICAÇÃO

Como referimos anteriormente, a compressão da informação de forma não destrutiva só é possível devido à redundância (informação repetida, que pode ser escrita de forma reduzida, recorrendo a uma forma de representação diferente) e dependência estatística (o símbolo seguinte está estatisticamente ligado ao símbolo anterior, por exemplo, numa imagem se um pixel está a preto, é muito provável que o seguinte também esteja).

Hoje em dia existem diversos algoritmos de compressão não destrutiva e, aquando da escolha de qual usar, deve ter-se em consideração múltiplos fatores, nomeadamente: taxa de compressão e tempo de compressão/tempo de descompressão.

As fórmulas de cálculo de cada um encontram-se explicitadas abaixo:

$$Compression\ Ratio = \frac{Uncompressed\ file\ size}{Compressed\ file\ size}$$

$$Decompression\ Speed = \frac{Compressed\ file\ size}{Time\ to\ decompress}$$

$$Compression\ Speed = \frac{Uncompressed\ file\ size}{Time\ to\ compress}$$

Taxas de compressão mais elevadas, normalmente levam a um maior uso da capacidade de processamento dos dispositivos que são usados quer para a compressão, quer para a descompressão. Assim sendo, também o tipo de dispositivos utilizados pesam na escolha do algoritmo de compressão a utilizar.

Na compressão dos dados são percorridas várias etapas sendo utilizados vários algoritmos em cada uma delas, algoritmos esses que vão ser mais detalhados na próxima parte. Mas, de uma forma geral a primeira etapa consiste na transformação da fonte através de algoritmos como os Predictors e o

BW (Burrows - Wheeler). Através desta transformação conseguimos estabelecer uma ordem na informação fornecida o que vai melhorar a eficiência das etapas seguintes. A segunda etapa consiste na análise da dependência estatística e tratamento dos dados gerando agrupamentos e tirando partido da redundância da fonte explorada anteriormente. São utilizadas técnicas como a codificação por dicionário, modelação através de cadeias de Markov e algoritmos como o Run Length Encoding. Com estes agrupamentos já obtidos podemos partir então para a etapa final! A etapa final corresponde à representação desses agrupamentos utilizando técnicas de codificação entrópica como por exemplo codificação aritmética e árvores de Huffman que se servem dos agrupamentos gerados anteriormente e da sua probabilidade para os melhor representar servindo-se do menor número de bits possível reduzindo assim o espaço ocupado pela informação original.

III. ALGORITMOS USADOS

1ª Etapa: Redundância da Fonte

Preditors

Este tipo de algoritmos tem como objectivo fazer uma transformação da fonte através de previsões e de relações entre elementos adjacentes da mesma fonte. Algoritmos como os de Delta Encoding são exemplo de bons Preditors. De forma sucinta o Delta Encoding consiste em exprimir uma fonte de dados através das semelhanças que elementos têm com os anteriores ou seja olhando para os dados e exprimindo-os através de variações.

BW (Burrows -Wheeler)

Este algoritmo é utilizado para aumentar a redundância da fonte de informação.

Funcionamento do algoritmo:

- S é a fonte de informação, que contém um caracter cuja função é marcar o final de S (EOF), S tem tamanho n (algumas versões deste algoritmo já não precisam do EOF).
- S é submetida a um total de n rotações (reversões) obtemos, assim, uma matriz de strings. Depois a matriz n por n com as strings obtidas a partir de S são ordenadas por ordem lexical ou numérica.
- Entre todas as linhas dessa matriz existirá uma igual a S. Selecionamos então a última coluna da matriz ordenada e o índice da string da matriz igual a S, o que nos permite reverter o processo posteriormente.

2ª Etapa: Dependência estatística

LZ77

Este algoritmo utiliza umas search window, de tamanho n (2 KiB, 4 KiB ou 32 KiB por exemplo), nesta são mantidos os últimos n símbolos enviados/recebidos e uma look-ahead window de determinado tamanho.

- Aquando do envio de mais símbolos, é efetuada uma procura na janela pela maior sequência começada pelo símbolo da sequência a enviar, a maior correspondência é então codificada da seguinte forma {<offset>, <length>, <next_symbol>}.

Offset: número de símbolos que têm de ser percorridos para trás na janela deslizante até chegar ao símbolo onde é começada a cópia.

Length: Número de símbolos a ser copiado da janela deslizante;

Next_symbol: O próximo símbolo que tem de ser enviado e que não se encontrava presente na sequência copiada da janela deslizante.

- Assim, dependendo do tamanho da janela a ser copiada, esta codificação pode ser deveras eficiente no envio de informação.
- Se não for encontrada uma sequência para copiar no search window, o offset e a length são 0,diminuindo a eficiência do algoritmo

LZW

LZW é uma versão do LZ78, recorrendo também a um dicionário. Um dos diferenciais do LZW para o LZ78 está na maneira como o dicionário é inicializado, no LZW este é inicializado com os caracteres do alfabeto da fonte, enquanto que no LZ78 este encontra-se vazio.

O dicionário é construído de forma adaptativa da seguinte forma:

- Procura-se no dicionário qual é a maior cadeia que mais caracteres têm iguais ao que se pretende enviar;
- Envia-se o índice do dicionário respectivo à melhor correspondência;
- Adiciona-se mais uma entrada no dicionário com o código da mensagem enviada + caracter seguinte da fonte de informação.

RLE

Run length encoding, consiste em reduzir uma sequência de símbolos iguais a uma sequência de 3 símbolos. Desses 3 símbolos fazem parte o símbolo que se repete na fonte de informação, o número de vezes que o símbolo é repetido e um símbolo especial que permite diferenciar esta forma de comprimir informação de uma sequência idêntica na fonte que em nada tem a ver com o reduzir de uma sequência.

3ª Etapa: Codificação Entrópica

Huffman Coding

Algoritmo que tira partido da probabilidade de ocorrência dos símbolos da fonte de informação, atribuindo aos que mais se repetem códigos de

comprimento menor, enquanto que os que menos se repetem são codificados com códigos mais extensos.

Os códigos são obtidos por via da construção de uma árvore a partir das folhas agrupando sempre os que têm menor probabilidade e, quando há mais do que dois com a mesma probabilidade e esta é a mais baixa, escolhem-se os dois que estão mais longe da raiz (assim garantimos que a variância dos códigos é mínima). Depois, continua-se este raciocínio até que a probabilidade depois da soma dos nodos dê 1.

Os códigos resultantes são instantâneos e unicamente decodificáveis (códigos de prefixo).

Arithmetic Coding

Este algoritmo codifica a informação com base na probabilidade de cada símbolo na fonte de informação.

Para codificar o primeiro símbolo da fonte de informação divide-se o intervalo [0, 1] pelos símbolos do alfabeto da fonte (com base na probabilidade de cada um), para o segundo símbolo dividem-se todos os subintervalos criados para o símbolo anterior com base nas probabilidades de cada símbolo e assim sucessivamente, até termos um intervalo que corresponda à fonte a enviar. Depois, basta escolher um valor que pertença a esse intervalo para enviar a fonte, normalmente escolhe-se o valor médio do intervalo.

Ainda são utilizados outros algoritmos de compressão que não vamos detalhar. O LZMA e o PPMd são dois desses algoritmos. O primeiro é um algoritmo da família Lempel-Ziv que utiliza modelação de Markov na codificação do elementos do dicionário, o segundo utiliza também esta modelação fazendo uma previsão probabilística através de uma cadeia de ordem n.

IV. COMBINAÇÕES MAIS HABITUAIS

1. Deflate

Usando o Deflate a compressão é alcançada em duas etapas principais: associação e substituição de séries de bytes (usando para o efeito o LZ77 com janela deslizante de 32KB e buffer de look-ahead de 258 bytes) e ainda substituição de símbolos tendo em conta a sua frequência de uso (usando codificação de Huffman).

O Deflate consiste numa série de blocos com um cabeçalho de 3 bits. O primeiro bit dá informação sobre se existem (bit a 1) ou não (bit a 0) mais blocos a seguir. O segundo e terceiro bit, por outro lado, dão informação sobre o tipo de codificação usado no bloco. Recebendo 00 representa um bloco que contém uma sequência de bits (que não foram repetidos) com tamanho entre 0 e 65.535 bytes, 01 representa um bloco codificado usando uma árvore de Huffman pré acordada, 10 representa um bloco comprimido com uma tabela de Huffman e 11 representa um bloco reservado que não pode ser usado.

2. PNG

O processo de compressão PNG envolve 2 partes, sendo elas: pré compressão (usando Predictor) e a conversão propriamente dita (usando Deflate). Na pré compressão da imagem é usado um *filter method* que é aplicado a toda a imagem e ainda um *filter type* que é aplicado a cada uma das linhas. O *filter method 0* (único método atualmente definido para o PNG) prevê a cor de um pixel com base nas cores dos pixels ao seu redor e subtrai ao valor real do pixel o valor previsto. Na compressão propriamente dita é usado o Deflate com uma janela de 32KB. Na compactação usando deflate pode ser utilizado um algoritmo de Huffman de comprimentos fixos ou adaptativo (personalizado) sobre os blocos de dados gerados pela codificação anterior através de LZ77.

V. ANÁLISE DE DADOS

Após o estudo dos vários algoritmos e das combinações mais usuais chegamos a conclusão que alguns algoritmos têm vantagens em termos de velocidade de compressão como por exemplo o Deflate mas que têm défices no rácio de compressão. Outros algoritmos como os da família Lempel-Ziv (LZMA que deriva do LZ77 por exemplo) atingem rácios de compressão e velocidades bastante bons o que os tornam bastante versáteis. E algoritmos como os da família PPmd atingem uma elevada compressão mas são muito lentos e exigem bastantes recursos de uma máquina. Como em tudo cada caso é um caso e temos de escolher o algoritmo que nos traz maior vantagem para a situação em que o queremos aplicar. Reparamos que os algoritmos são utilizados de uma forma encadeada para que se consiga obter melhores resultados.

Algoritmo de compressão de dados	Taxa de compressão (média)	Velocidade de compressão (média) MB/s	Velocidade de descompressão (média) MB/s
Deflate	3.825	9.732	91.342
LZMA	5.88	1.669	57.742
PPMd	6.095	6.457	5.748

Tab1. Aplicação destes algoritmos no Silesia Corpus

Neste trabalho experimentamos também algumas combinações diferentes com o intuito de obter uma taxa de compressão média o mais baixa possível. Recorrendo para o efeito a algumas estratégias pré definidas.

VI. POSSÍVEL SOLUÇÃO PARA COMPRESSÃO NÃO DESTRUTIVA

Considerações iniciais e Dataset

Após explorarmos alguns dos vários algoritmos utilizados hoje em dia para comprimir dados de forma não destrutiva desenvolvemos uma investigação centrada na tentativa de obter um codec de compressão que sendo baseado em alguns destes métodos explorados nos permitisse de forma eficiente obter a maior taxa de compressão possível e ao mesmo tempo fazê-lo em tempo útil evitando como sempre o gasto excessivo de memória no processo de encoding. Tentamos então encadear alguns algoritmos já descritos anteriormente que estão presentes nas diversas etapas de codificação ou que englobam em si combinações de algoritmos das diversas fases. Experimentamos então algoritmos como o Burrows -Wheeler e o RLE que caem sob a categoria de preditores e algoritmos que são combinações destes preditores com algoritmos das etapas de dependência estatística e de codificação entrópica como por exemplo o ppm o bzip2 e o png.

Analisamos os resultados obtidos na individualidade de todos os algoritmos e tentamos construir uma sequência lógica numa tentativa de encontrar um codec que fosse bom para o dataset fornecido para teste. O dataset é composto por dois ficheiros sendo o primeiro documento/livro em formato (.txt) e outro uma imagem monocromática em formato (.bmp).

Detalhes sobre o dataset são apresentados na tabela abaixo:

File Name	Data type	Size (Bytes)	Description
war_and_peace	Text in english (.txt) format	3 359 549	The Project Gutenberg EBook of War and Peace, by Leo Tolstoy
cromenco_c10	Image File (.bmp) format	33 987 318	Monochromatic image of a chromenco computer and other devices

Tab2. Dataset fornecido para a recolha de dados

CODEC conceptualizado e suas componentes

Como referido anteriormente através do uso de alguns algoritmos explorados anteriormente e outros que acabamos por aplicar após verificar que poderiam ser candidatos a ter um lugar na nossa solução pela sua eficiência rapidez e sobretudo elevada taxa de compressão como o caso do Bzip2 que é composto por 9 algoritmos das diferentes etapas de compressão e que se provou um excelente aliado na nossa tentativa de encontrar um codec que garanta a compressão ótima deste dataset.

O codec que construímos encontra-se detalhado no esquema seguinte:

Texto

Imagem

1º Etapa : Burrows - Wheeler (e base64)

Como podemos ver no esquema começamos por aplicar à fonte de dados o algoritmo burrows -Wheeler. A sua escolha na nossa escolha de sua implementação advém do facto de esta transformada facilitar a compressão por um algoritmo que venha a seguir na nossa sequência. Através do seu uso conseguimos organizar a fonte de dados agrupando os seus símbolos o que nos permite obter explorar a redundância da fonte aumentando-a como já vimos anteriormente. Este algoritmo aplica-se com grande eficiência em fontes de texto e produz bons resultados no entanto em imagem fomos forçados a construir uma solução de engenharia que nos permitisse aplicar um algoritmo concebido para fontes textuais. Daí na solução para imagem o nosso codec apresentar um passo extra no início do codec que se relaciona com o tratamento dos dados e que iremos explicar a seguir.

Na aplicação deste algoritmo em imagem deparamo-nos com um problema. A transformada foi construída e pensada para atuar em fontes de dados textuais e não em imagem pelo que para aplicar o algoritmo teríamos de encontrar uma forma de converter os dados em binário lidos aquando a abertura da imagem para dados sobre os quais o algoritmo pudesse operar. Chegamos à conclusão de que uma solução possível seria através do uso de um processo de codificação que nos permitisse representar esses caracteres binários como sendo strings sobre as quais o algoritmo pudesse trabalhar. Utilizamos então para isso um processo de codificação que é bastante utilizado hoje em dia na internet e em dispositivos que tem que tratar dados binários e que apenas foram concebidos para trabalhar dados textuais e aplicamos esse processo no nosso problema resolvendo-o. O método de codificação é denominado *Base64*.

Base64

Permite representar informação como sequências de caracteres encontrados numa tabela

com 64 caracteres diferentes (estes 64 caracteres costumam ser os mais comuns entre todos os tipos de codificação e possíveis de ser imprimidos). Cada caracter da tabela é codificado com apenas 6 bits, em vez de 8 bits (byte).

- A fonte de informação é vista como bits (esta está agrupada em conjuntos de 8 bits, ou seja, bytes);
- Criam-se agrupamentos de 6 bits a começar pela esquerda;
- Verifica-se a correspondência desses 6 bits com um caracter da tabela;
- Escreve-se esse caráter na mensagem a ser transmitida;

Assim, consegue mandar-se todo o tipo de informação por canais que apenas suportam texto.

Utilizado nos serviços de email, permitem o envio de imagens pelos mesmos canais que o corpo do email, mesmo esta apresentando valores não possíveis de ser transmitidos pelo canal.

No nosso caso permite-nos tratar uma imagem como texto.

Conseguimos em suma aumentar a redundância da fonte através da sua aplicação a um texto e a uma imagem o que se torna bastante útil para o algoritmo que é utilizado na etapa seguinte. No entanto acabamos por introduzir algum overhead no ficheiro pelo facto de termos de guardar também o índice da matriz de rotações da string codificada através deste processo.

2º Etapa : Run Length Encoding (RLE)

Nesta segunda etapa pensamos utilizar o algoritmo RLE visto que irá pôr ao uso a redundância criada pela transformada Burrows -Wheeler e eliminar o overhead gerado por esta. Após a aplicação deste algoritmo as fontes de texto obtidas na etapa anterior e que apresentam conjuntos de símbolos iguais de forma sequencial conseguimos não só eliminar o overhead produzido pela aplicação do algoritmo anterior mas também comprimir a fonte de dados como pretendemos como objetivo do algoritmo. Assim provamos que os dados redundantes que foram gerados são assim colmatados através do uso deste algoritmo em conjunção com o anterior razão pela qual o incluímos no nosso codec nesta sequência.

Em suma através desta combinação de preditor até agora tiramos partido da distribuição redundante da fonte e conseguimos alcançar algo que se pode tornar vantajoso para a etapa seguinte.

Através do Burrows - Wheeler aumentamos a redundância da fonte e com o RLE tiramos proveito desta!

3º Etapa : Bzip2

Na terceira etapa pensamos em usar um algoritmo que se provasse poderoso e eficiente na compressão de texto e imagem. Estudamos portanto várias opções que funcionam tanto com texto como imagem (ppm e bzip2) e opções desenhadas apenas para imagem (png). Acabamos por optar por utilizar o bzip2 que por si só é a combinação de vários algoritmos alguns deles já utilizados anteriormente visto que se provou como sendo o algoritmo que nos permitia atingir maior taxa de compressão no dataset que estamos a utilizar como referência.

O bzip2 é um algoritmo que utiliza vários métodos explorados anteriormente na seguinte sequência:

$RLE \rightarrow BW \rightarrow MTF \rightarrow RLE \rightarrow Huffman \rightarrow \Delta$ *encoding*

O Bzip2 é um algoritmo de compressão que por si só já explora muito bem a redundância da fonte de informação, visto que utiliza os algoritmos Run Length Encoding, Move to Front e Burrows Wheeler, não exatamente por esta ordem. O algoritmo Move to Front, tal como o Burrows Wheeler, é utilizado com o intuito de aumentar a redundância da fonte de informação. A utilização destes algoritmos é precedida e seguida pela utilização do algoritmo Run Length Encoding, que tem como função tirar partido da redundância gerada pelos algoritmos referenciados anteriormente, levando à diminuição, em tamanho, da informação (isto nem sempre se verifica, está dependente da fonte de informação utilizada).

Sabendo que o Bzip2 foi desenvolvido de forma a tirar partido da redundância da fonte, pensámos em tentar aumentar a redundância desta antes de aplicar o algoritmo Bzip2. Assim sendo submetemos a fonte ao algoritmo Burrows Wheeler e só depois utilizámos o Bzip2.

Desta forma chegamos ao fim da hipótese de um possível codec para comprimir texto e imagem de forma não destrutiva. Agora iremos passar à apresentação de resultados que obtivemos durante a nossa investigação que serviram de apoio na secção seguinte fazendo também um comentário, retirando conclusões que nos permitam perceber a qualidade da nossa solução e ao mesmo tempo retirar conclusões para um futuro trabalho a desenvolver na área da compressão de texto e imagem.

VII. RESULTADOS DA INVESTIGAÇÃO

Para obtermos os resultados que nos permitissem ter algum feedback do trabalho desenvolvido servimo-nos da linguagem de programação python usando a mais estável na atualidade a versão python3.7. Através da wikipédia, github entre outros sites adquirimos algum código que tentamos adaptar e corrigir por forma a servir os nossos interesses. Também utilizamos algumas

livrarias standard de python que já trazem alguns algoritmos implementados como por exemplo o bzip2 (presente na livraria bz2), o png (presente na livraria de imagem Pillow) e o base64 (presente na livraria base64). Construímos funções nesta linguagem que nos permitem utilizar os diversos algoritmos e juntar estes de forma lógica através do acoplamento por ficheiro. Não temos quaisquer direitos de autor sobre o código que adquirimos online estando este sob uma open license o que nos permite o seu uso legal. Todo o código utilizado é disponibilizado juntamente com este documento.

Os resultados obtidos foram benchmarked numa máquina com:

Intel i7-8750H (12 cores) processador (clock speed de 2.200GHz), 16GB de RAM e correndo o sistema operativo Linux (Ubuntu 19.10 x86_64)

Algorithm	Text Original Size (bytes)	Text Final Size (bytes)	Ratio (txt)	Image Original Size (bytes)	Image Final Size (bytes)	Ratio (img)
Bzip2	3 359 549	8 884 54	3,78	33 987 318	7 717 164	4,40
PPM	3 359 549	14 398 61	2,33	33 987 318	8 205 563	4,14
PNG	———	———	——	33 987 318	12 772 549	2,66
BW	3 359 549	3 473 091	0,96	33 987 318	45 729 625	0,74
BW→RLE	3 359 549	3 456 008	0,97	33 987 318	32 302 300	1,05

Tab3.Resultados obtidos após aplicação direta destes algoritmos no Dataset fornecido

Após aplicação direta de alguns dos algoritmos verificamos desde logo algumas particularidades:

- O algoritmo Bzip2 quando aplicado diretamente nas fontes de dados fornecidas permite-nos obter as melhores taxas de compressão. As melhores taxas a seguir a este são obtidas pelo ppm e pelo png nesta ordem.
- O png sendo um algoritmo vocacionado para imagem apenas tem uma pior performance que o bzip2 e que o ppm.
- A aplicação da transformada Burrows - Wheeler não nos permite obter nenhum ganho em termo de taxa de compressão o que se justifica pelo facto de associado a este algoritmo existir um overhead no guardar de informação extra necessária para a posterior descodificação. Além disso o objetivo deste algoritmo é aumentar a redundância da fonte através da sua transformação pelo que não irá comprimir de forma nenhuma os dados presentes na mesma
- A nossa aplicação posterior do RLE com o intuito de lucrar com a transformação BW surtiu algum efeito o que é visível na imagem e comprova assim de certa forma a hipótese que tínhamos.

Testamos várias combinações dos algoritmos falados anteriormente na tentativa de verificar quais combinações potenciam uma maior compressão.

Os dados obtidos encontram-se registados na tabela abaixo.

Algorithm	Text Original Size (bytes)	Text Final Size (bytes)	Ratio (txt)	Image Original Size (bytes)	Image Final Size (bytes)	Ratio (img)
BW→RLE→ PPM	3 359 549	1 965 284	1.70	33 987 318	18 893 541	1.80
PPM→Bzip2	3 359 549	1 446 760	2.32	33 987 318	10 939 387	3.11
PNG→Bzip2	———	———	——	33 987 318	12 272 731	2.77
PNG→PPM	———	———	——	33 987 318	12 234 415	2.78
BW→Bzip2	3 359 549	1 954 283	1.72	33 987 318	17 342 202	1.96
Bzip2→PPM	3 359 549	944 779	4.37	33 987 318	7 785 234	3.55

Tab4.Resultados obtidos após aplicação de algumas combinações dos algoritmos anteriores no Dataset fornecido

Após a aplicação de vários algoritmos à fonte de dados de uma forma sequencial verificamos desde logo que o uso de alguns destes em conjunto compromete a qualidade de compressão acabando nós por verificar que a eficiência destes foi reduzida drasticamente comparando com o algoritmo original quando aplicado individualmente.

Também verificamos que em certos casos como nas sequências Bzip2→PPM e PPM→Bzip2 o simples trocar da ordem da aplicação dos mesmo produz efeitos notáveis na taxa de compressão conseguida o que de certa forma sugere que a ordem

de aplicação é deveras importante pode tornar-se um fator determinante na qualidade dos algoritmos. Na aplicação destes algoritmos tivemos que ajustar a sua eficiência devido às limitações da máquina visto que alguns destes se demonstraram muito exigentes em termos de memória como por exemplo o PPM. Este algoritmo quando aplicado na imagem usando um modelo de markov de ordem 3 provou ter uma utilização exponencial em termos de memória e uma complexidade algorítmica de $O(256^n)$ pelo que quando usado com imagem foi necessário utilizar um modelo de ordem 1 para garantir a obtenção de resultados.

Agora apresentamos na tabela seguinte os resultados obtidos após a aplicação do nosso algoritmo no dataset fornecido e algumas conclusões sobre os resultados obtidos. São apresentadas nas várias linhas da tabela os resultados parciais obtidos em texto e em imagem.

Algorithm Steps	Text Start Size (bytes)	Text Final Size (bytes)	Ratio (txt)	Image Start Size (bytes)	Image Final Size (bytes)	Ratio (img)
BW	**3 359 549**	3 473 091	0.96	**33 987 318**	45 729 625	0.74
BW→ RLE	3 473 091	3 456 008	0.97	45 729 625	32 302 300	1.05
BW→RLE → Bzip2	3 456 008	**1 952 669**	**1.72**	32 302 300	**17 502 354**	**1.94**

Tab5 .Resultados parciais e finais obtidos após aplicação do nosso algoritmo no Dataset fornecido

Apresentamos assim os resultados obtidos e a performance do nosso algoritmo. Realçado no gráfico estão os tamanho inicial e final dos ficheiros do dataset e as taxa de compressão finais obtidas em texto e imagem. Na obtenção de resultados o nosso algoritmo provou ser relativamente rápido apenas demorando mais tempo na compressão de imagem e na fase de codificação através da transformada Burrows - Wheeler. Em geral não conseguimos um resultado ótimo para a taxa de compressão mas mesmo assim um resultado satisfatório.

VIII. CONCLUSÃO E CONSIDERAÇÕES FINAIS

Concluímos que a nossa tentativa de aumentar a redundância, mesmo não tenha sido totalmente bem sucedida na sua função, levava a uma diminuição da taxa de compressão do algoritmo, ficando o ficheiro comprimido com maior tamanho usando o Burrows Wheeler antes do Bzip2 do que utilizando apenas o Bzip2.

Pensámos que isto poderia estar a acontecer devido ao aumento da fonte de informação passada para o Bzip2. O aumento da fonte de informação deve-se à utilização do Burrows Wheeler, que, ao reorganizar a fonte, adiciona o índice da matriz de rotação que contém o bloco reordenado na forma correta. Os dados adicionados ao ficheiro não estão dependentes dos dados na fonte inicial, assim sendo é difícil prever padrões na "nova" fonte, podendo levar a um mau desempenho por parte do Bzip2.

Na compressão de imagem ao utilizarmos a codificação base64 identificamos também um possível fator que pode comprometer a eficiência dos algoritmos seguintes nomeadamente o Bzip2. Pensamos que possa justificar a fraca eficiência deste e a explicação encontrada para nós para tal prende-se com o facto de o algoritmo base64 tentar agrupar sequências de 8 bits em sequências de 6 bits ficando assim a mensagem de certa forma descalibrada, o que pode comprometer a eficiência dos algoritmos seguintes nomeadamente o RLE o que acaba por danificar a ação do Bzip2. Além disso o algoritmo base64 acaba por na sua ação aumentar o tamanho em bytes da fonte de dados o que pode vir a influenciar os resultados obtidos aquando o cálculo da taxa de compressão.

Para combater o aumento da fonte de informação passada ao Bzip2, introduzimos o algoritmo Run Length Encoding entre o Burrows Wheeler por nós adicionado e o Bzip2. Os resultados não foram exatamente os esperados, visto que levou a uma diminuição da taxa de compressão, em comparação com a obtida sem o RLE, na imagem.

No texto, a taxa de compressão aumentou ligeiramente. Para o ficheiro de texto utilizado a diferença não é muito significativa, mas para ficheiros maiores esta pode vir a aumentar e tornar-se algo a levar em consideração.

Partimos de um pensamento, que para nós está correto, que foi o de tentar aumentar a redundância da fonte e assim aumentar a taxa de compressão deste, o que não acabou por se verificar.

O nosso algoritmo apresenta resultados piores do que utilizar apenas o Bzip2, no entanto, após a realização destas experiências, ficámos com mais interesse no tema e com uma ideia do trabalho que está por detrás da criação de um algoritmo de compressão de informação, mesmo que muito reduzida. Pensamos também que a solução para este problema pode ser encontrada através da investigação de estratégias que nos permitam encadear os algoritmos sem comprometer a sua eficiência deixando assim esta sugestão para algum trabalho futuro que pegue nas ideias por nós desenvolvidas e exploradas aplicando-as melhorando

assim a taxa de compressão obtida e se possível a rapidez do nosso algoritmo

VI. BIBLIOGRAFIA

[1]https://en.wikipedia.org/wiki/Predictive_analytics

[2]https://en.wikipedia.org/wiki/Burrows–Wheeler_transform

[3]https://en.wikipedia.org/wiki/LZ77_and_LZ78

[4]https://en.wikipedia.org/wiki/Lempel–Ziv–Welch

[5]https://pt.wikipedia.org/wiki/Codificação_run-length

[6]https://en.wikipedia.org/wiki/Huffman_coding

[7]https://en.wikipedia.org/wiki/Arithmetic_coding

[8]https://en.wikipedia.org/wiki/Portable_Network_Graphics

[9]Modern Lossless Compression Techniques: Review,Comparison and Analysis

[9]https://en.wikipedia.org/wiki/Portable_Network_Graphics

[10]https://en.wikipedia.org/wiki/DEFLATE

[11]https://www.w3.org/TR/REC-png.pdf

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

[13]https://en.wikipedia.org/wiki/Prediction_by_partial_matching

[14]https://en.wikipedia.org/wiki/Base64

[15]https://stackabuse.com/run-length-encoding/

[16]https://github.com/nayuki/Reference-arithmetic-coding

[17]https://www.nayuki.io/page/reference-arithmetic-coding

[18]https://en.wikipedia.org/wiki/Bzip2

[19]https://docs.python.org/3/library/bz2.html

[20]https://docs.python.org/3/library/base64.html