Compressão de Arquivos FASTQ

Evolução do método de compressão LZW aplicado a propósitos específicos

Samuel Varela Melz

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2024

Samuel Varela Melz

Compressão de Arquivos FASTQ

Artigo apresentado ao curso de Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Derzu Omaia

Abril de 2024

Catalogação na publicação
Seção de Catalogação e Classificação

M532c Melz, Samuel Varela.
Compressão de arquivos FASTQ / Samuel Varela Melz. - João Pessoa, 2024.
21 f. : il.

Orientação: Derzu Omaia.
TCC (Graduação) - UFPB/CI.

1. FASTQ. 2. Compressão. I. Omaia. II. Derzu. III. Algorítimo. IV. Título.

UFPB/CI CDU 004.421

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado ***Compressão de Arquivos FASTQ*** de autoria de Samuel Varela Melz, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Derzu Omaia
CI/UFPB

Prof. Dr. Leonardo Vidal Batista
CI/UFPB

Ms. Annie Elisabeth Beltrão de Andrade
A3Data

João Pessoa, 13 de maio de 2024

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

Compressão de Arquivos FASTQ

Samuel V. Melz[1], Derzu Omaia[1]

[1]Centro de Informática – Universidade Federal da Paraíba (UFPB)

João Pessoa – PB – Brasil

samuel.melz@cc.ci.ufpb.br, derzu@ci.ufpb.br

Abstract. *This work introduces a purpose-specific data compression algorithm for FASTQ files, leveraging the known structure of these files as an improvement element for compression ratio. The chosen base method is Lempel-Ziv-Welch (LZW), a well-known and widely used lossless and contextual compression method. The information about the file was used to generate dictionaries more suited to each block of the file through preprocessing, opposing the usual method of generating a single dictionary, without considering the file type's particularities. Comparisons were made with the Rar and Zip methods, and the proposed method achieved higher compression rates.*

Resumo. *Este trabalho apresenta um algoritmo de compressão de dados de propósito específico, para arquivos do tipo FASTQ, utilizando-se da estrutura já conhecida deste arquivo como um elemento de melhoria da razão de compressão. O método base escolhido foi o Lempel-Ziv-Welch (LZW), método de compressão já conhecido e difundido, sem perdas e contextual. Foram utilizadas as informações sobre o arquivo para, através de um pré-processamento, gerar dicionários mais adequados a cada bloco do arquivo, em contraposição ao método usual de gerar um único dicionário, sem levar em consideração as particularidades do tipo de arquivo. Foram realizadas comparações com os métodos Rar e Zip, e o método proposto obteve taxas de compressão superiores.*

1. Introdução

Com o aumento da capacidade computacional ocorrida ao longo das últimas décadas, surgiram também novos usos para os computadores, como a análise de sequências genéticas. Dentre os projetos que lidaram com isso, um dos mais conhecidos é o Projeto Genoma Humano (Collins, Morgan, Patrinos, 2003). Com o passar dos anos, esse campo de estudo cresceu e tornou-se mais acessível à população de forma geral. Hoje, há diversas empresas que trabalham com análise de DNA espalhadas pelo mundo, seja para análise de ancestralidade, seja para análise de propensão a doenças, alergias etc. Empresas desse tipo necessitam de uma grande capacidade de armazenamento para armazenar amostras genéticas em larga escala de diversas pessoas, não podendo se desfazer delas, pois são o ponto chave de seu negócio. Com outras aplicações, mas nesse sentido, surgiram ao longo da história diversos métodos de compressão, para diferentes usos, como por exemplo o algoritmo de Huffman, LZ77, Deflate e PPM para textos; JPEG, RLE e DCT para imagens, dentre outros para diferentes formatos (Salomon, 2007). Esses algoritmos de compressão podem ser divididos em dois tipos: com perdas e sem perdas. Como estamos tratando de análises de DNA, cada informação é importante, não podemos trabalhar com uma

margem de erro aceitável, pois a mínima alteração pode comprometer toda a amostra. Usualmente, para essa finalidade, temos os algoritmos rar e zip que são bastante eficientes, de propósito geral e sem perdas. Dessa forma, busca-se neste trabalho, desenvolver um algoritmo de propósito específico para arquivos genéticos do tipo FASTQ (Cock, 2010), usualmente utilizados para estudos no campo genético - devido a sua simplicidade e facilidade de leitura humana, gerando assim diminuição no espaço utilizado para armazenamento. Buscamos também, analisar esses métodos, entendendo seu funcionamento e gerando alterações no método original que possam trazer algum tipo de vantagem, para o uso geral ou para o uso específico, gerando um acréscimo a esse campo de estudo, que possa, além de ser objeto de estudo futuro, também ser utilizado de forma prática.

1.1. Objetivos

1.1.1. Objetivo principal

Desenvolver um algoritmo de propósito específico, tendo por base o método FASTQ, que supere, na compressão deste tipo de arquivo, os métodos rar e zip, tomados neste artigo como balizadores.

1.1.2. Objetivos específicos

Análise do método e do tipo de arquivo, de forma a buscar casos de maior possibilidade de ocorrência de padrões.

Demonstrar a possibilidade de uso prático do método proposto, indicando possíveis melhorias para estudos e implementações futuras.

2. Trabalhos Relacionados

2.1. Árvore de códigos binários

A compressão de dados genéticos é um tema bastante recorrente em artigos acadêmicos, o que demonstra a utilidade e necessidade desse estudo. Um trabalho que segue uma linha próxima deste foi realizado por Nishad PM e R. Manicka Chezian em um pós-doutorado na Índia (Nishad, Chezian, 2012).

Aproveitando-se da baixa variabilidade dos nucleotídeos, decidiram utilizar apenas dois bits para identificar cada um deles: ‘00’, 01’, ‘10’ e ‘11’. Dessa forma, cada byte era capaz de representar uma sequência de 4 nucleotídeos. A partir de então, as posições no dicionário são usadas para armazenar bytes inteiros (Nishad, Chezian, 2012).

Ao fim desse processo, ocorre a criação de uma árvore binária, onde uma determinada posição do dicionário é convertida em um número binário pelo caminho percorrido da raiz até as folhas. Dessa forma, a menor sequência gerada pela árvore contém 3 bits. O valor gerado por essa árvore para a posição do dicionário gera o valor que será efetivamente escrito no arquivo final (Nishad, Chezian, 2012).

O método apresenta um excelente resultado para o trecho genético, com uma taxa de compressão de aproximadamente 94%. Embora tenha sido um ótimo resultado, seria importante analisar a possibilidade de melhoria trocando a construção da árvore binária como é feita, pela

construção de uma árvore de Huffman. Como há uma diferença de tamanho entre as chaves do dicionário, não bastaria analisar o número de ocorrências, mas sim o produto entre as ocorrências e o comprimento da string salva naquela posição do dicionário.

2.2. MZPAQ

Outro método de compressão para arquivos do tipo fastq é o MZPAQ, desenvolvido por El Alalli e Arshad, que utiliza uma estratégia de pré-processamento parecida com o que foi desenvolvido neste trabalho (El Allali, Arshad, 2019).

O nome deste método surge através da combinação de métodos que o originam: MFCompress e ZPAQ. O primeiro passo do método ZPAQ é realizar um pré-processamento, dividindo o arquivo em linhas, que são os blocos unitários do arquivo FASTQ, conforme será explicado mais à frente. As duas primeiras linhas são comprimidas com o método MFCompress. A terceira linha é ignorada e reconstruída em tempo de execução, uma vez que contém apenas um caractere "+" e um comentário, não sendo essencial para a leitura computacional do arquivo. Já a quarta linha é comprimida com o método ZPAQ. Após a realização de cada compressão individualmente, ocorre a combinação desses sub-arquivos em um único arquivo binário (El Allali, Arshad, 2019).

A descompressão funciona de maneira simples, dividindo o arquivo binário novamente em sub-arquivos que irão ser individualmente descomprimidos, segundo suas linhas. A terceira linha, como dito anteriormente, não é guardada no arquivo comprimido, porém como se conhece o formato do arquivo, ela é recriada artificialmente com a sua única parte imprescindível - o caractere "+" . Em seguida, os arquivos são novamente combinados em um único arquivo (El Allali, Arshad, 2019).

Apesar de obter um bom resultado, excelente em alguns cenários, não é possível classificar esse método como um método de compressão sem perdas, uma vez que a terceira linha - de comentários - é perdida durante a compressão.

3. Referencial Teórico

3.1. Compressão de dados

Ao falarmos de computadores, tratamos de máquinas limitadas em recursos. Não temos capacidade computacional infinita, seja para armazenar, transmitir ou processar. Com o passar dos anos e o aumento dessas capacidades, o esperado é que houvesse uma folga maior entre os recursos utilizados pelo computador e as capacidades máximas destes, porém não é o que se viu acontecer. Com o aumento da capacidade computacional, houve também o aumento dos recursos que são utilizados: mais armazenamento, mais processamento, mais necessidade de que tudo ocorra de maneira mais eficiente (Deorowicz, Grabowski, 2013).

Dentre diversas técnicas que foram desenvolvidas ao longo do tempo a fim de gerar mais eficiência em diversas áreas, quando tratamos de tamanho de arquivos, desenvolveram-se métodos de compressão para que assim a mesma informação seja armazenada em menos espaço, transmitida com menor banda ou de forma mais rápida. Um exemplo disso é o aplicativo WhatsApp, que, como um aplicativo de mensagens, trabalha massivamente com a transmissão e armazenamento de dados. Com as funcionalidades implementadas, como envio de imagens e

vídeos, uma largura de banda maior é necessária para em tempo razoável entregar a informação ao destinatário. Para isso, as imagens, por exemplo, utilizam o método de compressão PNG, que gera a possibilidade de apresentar uma prévia, com baixa transferência de dados, antes que o usuário realmente a solicite. Quando ocorre a solicitação de transmissão, a imagem é enviada compactada e em seguida, no aparelho, descompactada (Anwar, 2021).

Os métodos de compressão podem ser divididos de acordo com seus usos ou características (Câmara, 2009), por exemplo:

- Quanto à integridade
  - Com perdas

    Ocorre normalmente em imagens, quando alguma variação de dados não traz grandes diferenças (Câmara, 2009).

  - Sem perdas

    Quando é necessário que se mantenha a integridade do arquivo original (Câmara, 2009).

- Quanto ao tipo de arquivo
  - Imagens

    Quando o arquivo a ser comprimido tem o formato de imagem, usualmente uma matriz de pixels (Fitriya, Purboyo, Prasati, 2017).

  - Texto

    Nesse caso, o formato do arquivo é uma sequência de caracteres (Fitriya, Purboyo, Prasati, 2017).

- Quanto ao tipo de compressão
  - Lógica

    Ocorre no projeto de representação de dados, como, por exemplo, datas que são representadas internamente como um número inteiro, como no Excel em que o número 1 representa a data 01/01/1900 (Câmara, 2009).

  - Física

    Ocorre a partir do arquivo já existente, através da repetição de caracteres ou sequências ou ainda da redução do espaço utilizado para armazenamento de caracteres (Câmara, 2009).

3.2. Algoritmo de Huffman

O algoritmo de Huffman é um algoritmo de substituição de valores que utiliza uma árvore binária para definir o código que será dado a cada caractere, sendo um algoritmo de fácil implementação e alta eficácia. Ocorre uma primeira leitura do arquivo, contando as ocorrências de cada caractere. Em seguida, estes são ordenados de forma decrescente em número de ocorrências e ocorre a criação de uma árvore binária, na qual, a partir do nó raiz, lê-se novamente o arquivo e o

caractere é substituído por uma sequência de bits indicada pelo caminho percorrido na árvore binária do nó raiz até o nó folha, que corresponde àquele caractere que será comprimido (Câmara, 2009).

Para a decodificação, lê-se cada caractere individualmente e de acordo com seu valor se percorre a árvore binária. Caso o nó ao qual o algoritmo avançou não seja um nó folha, o algoritmo continua lendo o arquivo, caso contrário o algoritmo substitui a sequência de bits pelo caractere encontrado (Câmara, 2009).

Como não há nenhum indicativo de que a sequência de determinado caractere chegou ao fim, é importante que uma sequência não seja subsequência de outra, gerando possíveis substituições incorretas. Nas sequências geradas, os símbolos de maior ocorrência recebem menos bits para decodificação (Câmara, 2009).

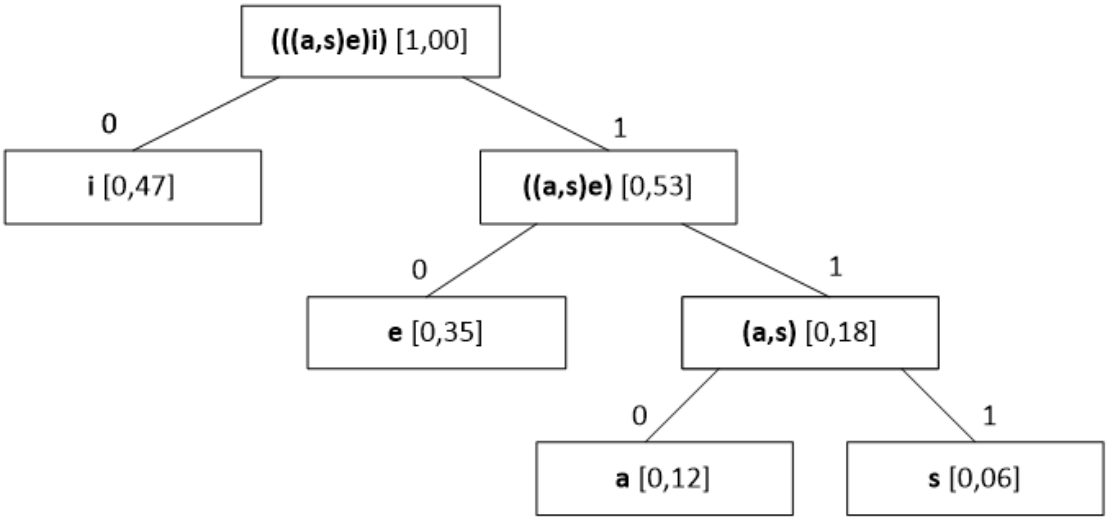

Figura 1. Exemplo de árvore gerada ao fim do algoritmo de Huffman

3.3. Algoritmos LZ

LZ77, LZ78 são métodos de compressão sem perdas baseados em dicionários desenvolvidos por Jacob Ziv e Abraham Lempel. Já o método LZW é uma evolução do método LZ78 introduzida por Terry Welch, que torna o método menos custoso computacionalmente. Embora todos trabalhem com dicionários de forma a substituir grandes sequências por sequências menores, há algumas diferenças fundamentais entre eles (Zeeh, 2003).

3.3.1. LZ77

O LZ77 consiste em um dicionário e uma janela deslizante (ou buffer) que avança pelo texto. O dicionário são os n caracteres anteriores decodificados, sendo n o tamanho do dicionário, logo, as posições anteriores são automaticamente eliminadas quando o dicionário avança pelo texto codificado. A ideia é buscar a maior sequência de caracteres que esteja presente no dicionário desde que não exceda o tamanho do buffer. Quando essa sequência é encontrada, o algoritmo converte a sequência original em um token ternário do tipo <a, b, c>. Neste caso, ao ler um símbolo, *a* representa a quantidade de posições que a janela deslizante precisa voltar para

encontrar a sequência já lida que está presente no dicionário, *b* indica o tamanho da sequência que será copiada e *c* indica o primeiro caractere diferente após a sequência, que agora se torna parte do dicionário (Zeeh, 2003).

A decodificação ocorre lendo esses tokens e, com o dicionário inicialmente vazio, acrescentando no output os caracteres de acordo com as informações passadas pelo token (Zeeh, 2003).

	sir␣sid␣eastman␣	⇒	(0,0,"s")
s	ir␣sid␣eastman␣e	⇒	(0,0,"i")
si	r␣sid␣eastman␣ea	⇒	(0,0,"r")
sir	␣sid␣eastman␣eas	⇒	(0,0,"␣")
sir␣	sid␣eastman␣easi	⇒	(4,2,"d")

Figura 2. Exemplo de uma codificação LZ77 e a saída gerada à direita

3.3.2. LZSS

Embora o método RAR seja proprietário, o que dificulta conhecer sua maneira de comprimir dados, sabe-se que em seu modo de execução geral, é utilizado o método LZSS, que é uma variação do método LZ77 (Salomon, 2007).

Há 3 pontos de melhoria, em relação ao método LZ77 tradicional (Salomon, 2007):

1. O buffer é mantido em uma fila circular.
2. O dicionário é tratado como uma árvore binária.
3. Os tokens são gerados com apenas dois campos.

O uso de uma árvore binária no dicionário permite que se faça uma busca de forma mais rápida, uma vez que o nó filho esquerdo é sempre menor que o nó atual e o nó filho direito é sempre maior que o nó atual. Como neste caso se trata de caracteres, usa-se a ordem da tabela ASCII para determinar quem é maior ou menor nessa classificação (Salomon, 2007).

Com relação ao token gerado, este contém apenas os valores *a* e *b* mostrados na seção 3.3.1. O valor *c*, que seria o valor seguinte à sequência, é tratado como início de outro token, evitando assim o gasto com esse valor que no LZ77 é tido apenas como um elemento fora da sequência (Salomon, 2007).

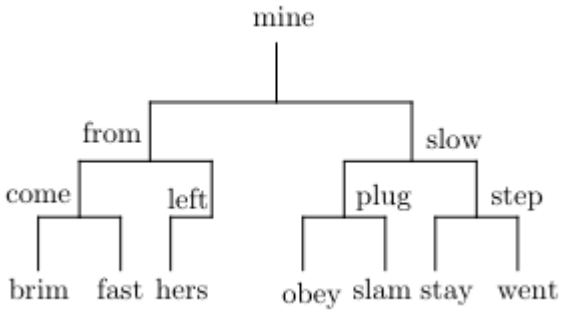

Figura 3. Exemplo de distribuição de palavras em um dicionário LZSS

3.3.3. LZ78

O algoritmo LZ78 consiste em um dicionário, no qual são inseridas as sequências de caracteres encontradas, dessa vez sem nenhuma janela deslizante, o que apresenta a vantagem de não eliminar nenhuma sequência já lida de acordo com o avançar do arquivo. O output também é um token, porém não mais contendo três posições, contendo agora apenas duas. O token é do tipo <a, b>, onde *a* é a posição da sequência no dicionário e *b* é o primeiro caractere após a sequência já existente no dicionário. Após inserir este token no output, a sequência *ab* é inserida no dicionário. A limitação desse método consiste no espaço de memória alocado para o dicionário que tende a apenas aumentar (Zeeh, 2003).

A decodificação ocorre com a leitura dos tokens gerados e a inserção das novas chaves no dicionário, sendo o valor delas concatenado ao caractere seguinte a fim de formar a sequência original (Zeeh, 2003).

“sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals”.

Dictionary		Token	Dictionary		Token
0	null				
1	“s”	(0,“s”)	8	“a”	(0,“a”)
2	“i”	(0,“i”)	9	“st”	(1,“t”)
3	“r”	(0,“r”)	10	“m”	(0,“m”)
4	“␣”	(0,“␣”)	11	“an”	(8,“n”)
5	“si”	(1,“i”)	12	“␣ea”	(7,“a”)
6	“d”	(0,“d”)	13	“sil”	(5,“l”)
7	“␣e”	(4,“e”)	14	“y”	(0,“y”)

Figura 4. Primeiras 14 inserções no dicionário para a frase acima, utilizando o método LZ78.

3.3.4. LZW

O algoritmo LZW utiliza a mesma ideia de dicionário do LZ78, porém não gera como saída um token. O valor de saída é apenas a posição da sequência no dicionário. Dessa forma, economiza-se espaço com separadores, utilizados nos métodos anteriores. O algoritmo percorre o arquivo lendo os caracteres, gerando sequências e buscando se esses valores existem no dicionário. Caso existam, toda a sequência é substituída pela posição da sequência no dicionário. O dicionário, inicialmente vazio ou preenchido com valores básicos, é acrescido da sequência encontrada + primeiro símbolo após a sequência, mas apenas a sequência já existente no dicionário é escrita no output (Zeeh, 2003).

Diferentemente de outros métodos de compressão sem perdas, como o algoritmo de Huffman, cada chave gerada para o arquivo codificado é de tamanho fixo, o que essencialmente limita a quantidade de chaves que se pode gerar. Ao utilizar-se 1 byte para cada caractere, o dicionário teria apenas 256 posições, o que geraria 256 sequências. Em um contexto normal,

onde há 256 caracteres básicos, isso implica gerar um arquivo exatamente igual ao arquivo de entrada, portanto é necessário gastar mais espaço por sequência, assumindo um uso pior de memória no começo, já que se usaria mais de um byte para um único caractere, o que leva ao fato de que para arquivos pequenos ou sem grandes repetições de caracteres, o método LZW pode gerar arquivos maiores que o original. Porém, em arquivos grandes o bastante, essa abordagem tende a fazer sentido pela quantidade de espaço economizado com sequências maiores no decorrer da compressão (Zeeh, 2003).

0	NULL	110	n	262	␣e	276	te
1	SOH	...		263	ea	277	eas
...		115	s	264	as	278	se
32	SP	116	t	265	st	279	es
...		...		266	tm	280	s␣
97	a	121	y	267	ma	281	␣se
98	b	...		268	an	282	ea␣
99	c	255	255	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
...		258	r␣	272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	sid	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Figura 5. Dicionário LZW com os caracteres básicos e as concatenações formadas ao longo do processo.

3.4. Algoritmo Deflate

Esse é o algoritmo utilizado pelo método ZIP, que consiste em uma combinação do método LZ77 com a árvore de Huffman. Ocorre primeiramente a codificação pelo método LZ77 e, em seguida, os caracteres isolados e as distâncias são codificadas através de uma árvore de Huffman. Já os valores de tamanho são codificados com uma segunda árvore de Huffman. É importante, ainda, dizer que esses valores e árvores não são únicos para todo o arquivo. O arquivo é dividido em blocos e o método é aplicado em cada bloco de forma a otimizar a compressão naquele trecho. Quando um bloco se torna muito grande e já não há um grande ganho de compressão, esse bloco é encerrado e se começa um novo (Oswal, Singh, Kumari, 2016).

Para a decodificação ou inflate, ocorre, para cada bloco, a decodificação da árvore de Huffman e em seguida a decodificação pelo LZ77, porém por questões de performance, existe a criação de tabelas que auxiliam a avançar na árvore de Huffman de forma acelerada (Oswal, Singh, Kumari, 2016).

3.5. Formato FASTQ

O FASTQ é um formato de arquivos genéticos, que além de simples, é de fácil entendimento para um leitor humano, tendo padrões de formação bem definidos. Ao analisar este formato de arquivo, podemos dividi-lo em 4 blocos que se repetem (Cock, 2010).

- Bloco 1

Este bloco, que começa com ‘@’, serve para identificação da sequência que vem a seguir. Além de indicar de qual subdivisão do código genético se trata, pode também conter alguma descrição (Cock, 2010).

- Bloco 2

Neste segundo bloco, encontramos a identificação das bases nitrogenadas lidas. É onde ocorre a maior repetição de dados, uma vez que se trata de apenas 5 possíveis valores: A(Adenina), T(Timina), G(Guanina), C(Citosina) e U(Uracila) (Cock, 2010).

- Bloco 3

O terceiro bloco consiste em um ‘+’ isolado ou seguido da mesma descrição após o ‘@’ do bloco 1 (Cock, 2010).

- Bloco 4

No último bloco, temos um caractere correspondente a cada caractere no bloco 2, pois se trata da qualidade da leitura dessas bases. Podem variar de 33 (‘!’), menor qualidade, a 126 (‘~’), maior qualidade (Cock, 2010).

Figura 6. Trecho retirado do arquivo SRR22399485_1.fastq (IHNA,2023)

4. Método Proposto/Metodologia

Conforme mencionado na introdução, os métodos ZIP e RAR são aqueles que buscamos vencer, por serem atualmente bastante eficientes e difundidos. Como a gênese desse trabalho ocorreu no componente curricular de Introdução à Teoria da Informação, decidiu-se continuar o que foi desenvolvido nele, utilizando o método LZW como método base, uma vez que já havia um grau de conhecimento sobre seu funcionamento, por ser um método aberto e que traz bons resultados.

4.1. Método utilizado

Sabendo como o arquivo funciona, percebeu-se que há um bloco em que há uma grande possibilidade de haver redução de tamanho, o bloco 2, visto que ele possui apenas 5 possibilidades de caracteres diferentes, o que levaria mais facilmente à formação de cadeias maiores no dicionário. Por esta razão, este bloco foi tratado de maneira diferente dos outros. Desta forma, dividimos o arquivo original em Bloco Geral, formado pelos blocos 1, 3 e 4, e Bloco Genético, formado apenas pelo bloco 2.

A partir dessa ideia, houve duas estratégias possíveis para a implementação da codificação:

1. Ler o arquivo e ao mesmo tempo codificar, alterando o dicionário e todas as variáveis a cada troca de bloco.
2. Realizar um pré-processamento, separando esses blocos, codificando cada um separadamente e reunindo novamente no mesmo arquivo final.

A estratégia 1 foi a primeira a ser utilizada, usando ponteiros para referenciar o dicionário correto e todas as variáveis a cada momento. Com o passar do tempo, não houve grande avanço devido à complexidade que o algoritmo estava tomando e resultados inesperados a todo momento, como invasões de valores que ocorriam entre os dicionários e entre outras variáveis, gerando valores que não deveriam estar naquele dicionário.

A partir de então, recomeçou-se tudo com a estratégia 2. Com ela, foi possível entender melhor o que estava acontecendo a cada momento do processo e assim corrigir erros, gerando um código mais limpo, revisável e inteligível.

4.2. Funcionamento

4.2.1. Compressão

Inicialmente, são criados três arquivos: o arquivo de destino, onde será escrito o arquivo original a ser comprimido, e dois arquivos auxiliares. Em seguida, o processo lê o arquivo original e o seu conteúdo é dividido entre os arquivos auxiliares. Um dos arquivos recebe todo o conteúdo do bloco 2, enquanto o outro arquivo recebe todo o conteúdo dos blocos 1, 3 e 4. Na tabela 1 é mostrada a proporção do arquivo original a que esses arquivos auxiliares correspondem.

Tabela 1 – Proporção entre o arquivo original e os arquivos auxiliares

Nome	Tamanho Original (kB)	Tamanho do arquivo - Bloco 2 (kB)	% Bloco 2	Tamanho do arquivo - Blocos 1, 3, 4 (kB)	% Blocos, 1, 3, 4
SRR2239948 5_1.fastq	2443540	1208423	49,5%	1235118	50,5%
SRR2239948 7_1.fastq	1297052	640865	49,4%	656187	50,6%
SRR2239949 0_1.fastq	3246652	1599632	49,3%	1647021	50,7%
SRR2239949 3_1.fastq	3102072	1536117	49,5%	1565955	50,5%

SRR2239949 4_1.fastq	2020479	1001949	49,6%	1018531	50,4%
SRR2239949 5_1.fastq	2383428	1180594	49,5%	1202835	50,5%
SRR2239949 6_1.fastq	3187427	1576229	49,5%	1611199	50,5%
SRR2239949 7_1.fastq	2504645	1237847	49,4%	1266799	50,6%
SRR2239949 8_1.fastq	2120186	1046819	49,4%	1073367	50,6%
SRR2239949 9_1.fastq	2807160	1386154	49,4%	1421006	50,6%
Média			49,45%		50,55%

Pode-se perceber, pela Tabela 1, que o bloco 2 ocupa aproximadamente metade (49%) do tamanho do arquivo original.

Ocorre, então, a inicialização dos valores que estarão presentes no dicionário do LZW. Neste método foi utilizado um dicionário fixo, com índices de 16 bits, ou 2 bytes, e não foram adotadas estratégias de exclusão de chaves para liberar espaço no dicionário.

Como não houve estratégias de liberação de espaço, é necessário explicar o funcionamento do método no caso de um valor inesperado para o bloco 2, como o N, que indica uma leitura imprecisa. Ao inicializar o dicionário com apenas 4 nucleotídeos e o byte SWITCH, teríamos o problema de não recuperar adequadamente o caractere N, porém, para este artigo, inicializamos todo o dicionário básico de 0 a 255. Poderia-se otimizar esse uso do dicionário com estratégias de exclusão caso surja o caractere N em algum momento.

Ao fim desse passo, realizamos a codificação LZW de um dos arquivos auxiliares, inserindo seu resultado no arquivo de destino com a adição de um byte especial, batizado como SWITCH, o qual ocupa um espaço no dicionário. Após este passo, todas as variáveis são reiniciadas, incluindo o dicionário, para que possa haver uma nova codificação LZW que seja eficaz para o segundo arquivo auxiliar. Essa nova codificação também utiliza índices de tamanho 16 bits no dicionário. Após essa nova codificação, temos a inserção do resultado no arquivo de destino, novamente acrescido de um byte SWITCH.

O byte SWITCH, tem como finalidade indicar a separação dos dois blocos que compõem o arquivo de destino, e também indicar o fim do arquivo. Esse byte foi criado, para que haja um ponto onde pode-se determinar de forma segura o fim do arquivo codificado e seu valor deve ser iniciado junto com o dicionário básico, impedindo assim que haja qualquer valor alocado com o SWITCH que usamos para controle. Dessa forma, é possível, no processo de descompressão, definir o começo e o fim de cada bloco do arquivo comprimido, permitindo assim que haja o tratamento individual de cada bloco pelas função de descompressão.

Ao fim desse processo, a estrutura do arquivo de destino é:

Quadro 1 – Formato do arquivo codificado

<arquivo_codificado_1><SWITCH> <arquivo_codificado_2><SWITCH>

Onde o arquivo_codificado_1 possui o bloco 2 codificado, e arquivo_codificado_2 possui os blocos 1, 3 e 4 codificados.

4.2.2. Descompressão

Na descompressão, novamente, são utilizados um arquivo de destino e dois arquivos auxiliares. Ocorre a criação ou truncamento dos arquivos, caso já existam, e o dicionário é inicializado de forma inversa ao que foi feito na codificação: se antes o dicionário retornava um valor inteiro para uma determinada sequência, agora ele retorna uma sequência para um determinado valor inteiro.

Conhecendo a estrutura do Quadro 1, a descompressão realiza a leitura sequencial do arquivo até encontrar o byte SWITCH, que marca o fim do <arquivo_codificado_1>. Em seguida, a função de decodificação é chamada recebendo <arquivo_codificado_1> como parâmetro e o resultado é escrito no arquivo auxiliar, Ocorre, então, a leitura do <arquivo_codificado_2>, que é decodificado e seu valor é escrito em outro arquivo auxiliar. Ao fim desse processo, os arquivos são recombinados, considerando a estrutura já conhecida do formato FASTQ. Terminado esse processo, o arquivo está reconstituído e idêntico ao original.

5. Resultados

Uma vez implementado o método, foram realizados os testes com os arquivos genéticos FASTQ produzidos pelo IHNA e os resultados foram comparados com os métodos que buscamos superar: rar e zip. De forma a manter o máximo de igualdade entre os métodos, foi limitado o tamanho do dicionário a 2 MB para o método proposto e para o rar, para o método zip não foi possível definir o tamanho máximo do dicionário. Neste artigo, adotou-se a Razão de Compressão como $RC = \frac{Tamanho\ Final}{Tamanho\ Original}$, portanto quanto menor o valor de RC, maior a compressão gerada (IHNA, 2023).

Tabela 2 – Resultados dos métodos de compressão

Nome do Arquivo	Tamanho Original (kB)	Tamanho após compressão - Zip (kB)	Razão de Compressão - Zip	Tamanho após Compressão - Rar (kB)	Razão de Compressão - Rar	Tamanho após compressão - método proposto (kB)	Razão de Compressão - método proposto
SRR22399485_1.fastq	2443540	1283226	0,5252	1271203	0,5202	1098226	0,4494
SRR22399487_1.fastq	1297052	680525	0,5247	673787	0,5195	583319	0,4497

SRR22399490_1.fastq	3246652	1546469	0,4763	1450248	0,4467	1288093	0,3967
SRR22399493_1.fastq	3102072	1503378	0,4846	1481719	0,4777	1255523	0,4047
SRR22399494_1.fastq	2020479	980411	0,4852	962244	0,4762	816229	0,4040
SRR22399495_1.fastq	2383428	1151799	0,4833	1117936	0,4690	961184	0,4033
SRR22399496_1.fastq	3187427	1540618	0,4833	1495169	0,4691	1286730	0,4037
SRR22399497_1.fastq	2504645	1215690	0,4854	1196511	0,4777	1015752	0,4055
SRR22399498_1.fastq	2120186	1028859	0,4853	1014930	0,4787	859934	0,4056
SRR22399499_1.fastq	2807160	1361912	0,4852	1342076	0,4781	1138205	0,4055
Média			0,4918		0,4813		0,4128

Como se pode atestar através da Tabela 2, o método proposto para a compressão, gera um resultado consideravelmente melhor que os métodos tradicionais, otimizando o uso de armazenamento, para esses arquivos, em aproximadamente 18% em relação aos outros. Desta forma, o arquivo SRR22399490_1.fastq, cujo tamanho é aproximadamente 3,24 GB, teria um tamanho comprimido de 1,28 GB no método proposto, enquanto que, no zip teria 1,54 GB e no rar 1,45 GB.

O entendimento do método LZW e da formatação do arquivo FASTQ permitiu que fosse encontrada uma maneira de alterar o arquivo original de forma a obter um aproveitamento melhor dos espaços no dicionário quando comparado ao fluxo normal do método, que ocorre de maneira cega.

Foi visto um grande aumento na capacidade de compressão, principalmente no bloco 2 do arquivo original, como era de fato esperado e como se pode constatar na Tabela 3, mostrando a grande efetividade que este pré-processamento gera no resultado final dos arquivos comprimidos.

Tabela 3 – Resultados da compressão para o bloco 2

Nome	Tamanho Original (kB)	Tamanho após compressão - método proposto (kB)	Razão de compressão - método proposto

SRR22399485_1.fastq	1208423	309642	0,2562
SRR22399487_1.fastq	640865	164219	0,2562
SRR22399490_1.fastq	1599632	403934	0,2525
SRR22399493_1.fastq	1536117	395362	0,2574
SRR22399494_1.fastq	1001949	257950	0,2574
SRR22399495_1.fastq	1180594	302090	0,2559
SRR22399496_1.fastq	1576229	403468	0,2560
SRR22399497_1.fastq	1237847	318737	0,2575
SRR22399498_1.fastq	1046819	269664	0,2576
SRR22399499_1.fastq	1386154	356830	0,2574
Média			0,2564

Como o arquivo que contém apenas o bloco 2 é um produto do método proposto, não se julgou conveniente explicitar a mesma comparação para os outros métodos, embora em alguns poucos testes realizados o desempenho tenha sido muito próximo entre o método proposto e a compressão com o RAR.

Outras métricas importantes para a escolha de um método dessa natureza são o tempo necessário para compressão e descompressão. Nas tabelas 4 e 5, foi feita a comparação do tempo de execução de cada método em segundos.

Tabela 4 – Tempo de compressão, em segundos, para cada método

Nome do Arquivo	Tempo de compressão - Zip	Tempo de compressão - Rar	Tempo de compressão - método proposto
SRR22399485_1.fastq	286	232	1451
SRR22399487_1.fastq	129	119	786
SRR22399490_1.fastq	326	424	2087
SRR22399493_1.fastq	285	400	1972
SRR22399494_1.fastq	153	262	1380
SRR22399495_1.fastq	199	325	1636
SRR22399496_1.fastq	287	433	2182
SRR22399497_1.fastq	248	341	1883
SRR22399498_1.fastq	235	291	1256
SRR22399499_1.fastq	244	374	1925
Média	239,2	320,1	1655,6

Tabela 5 – Tempo de descompressão, em segundos, para cada método

Nome do Arquivo	Tempo de descompressão - Zip	Tempo de descompressão - Rar	Tempo de descompressão - método proposto
SRR22399485_1.fastq	29	29	369
SRR22399487_1.fastq	16	15	199
SRR22399490_1.fastq	49	33	481

SRR22399493_1.fastq	41	34	471
SRR22399494_1.fastq	25	22	323
SRR22399495_1.fastq	35	30	357
SRR22399496_1.fastq	40	40	481
SRR22399497_1.fastq	42	29	364
SRR22399498_1.fastq	28	27	311
SRR22399499_1.fastq	45	33	405
Média	35	29,2	376,1

Pode-se notar que apesar de um aumento no grau de compressão obtido pelo método proposto, há um aumento de tempo médio, sendo de 592% em relação ao método zip e 417% em relação ao método rar. Para a descompressão, o mesmo efeito pode ser notado, sendo o aumento de 974% em relação ao zip e 1188% em relação ao rar.

O tempo de processamento do método proposto é muito alto, comparado aos das demais técnicas analisadas, contudo, o algoritmo desenvolvido não foi otimizado e ainda está em nível de protótipo, desta forma, ainda é passível de várias otimizações, como a paralelização.

Conforme descrito anteriormente, o método de compressão divide o arquivo original em dois. A partir desse ponto, não há necessidade de executar as compressões de forma sequencial, mas pode-se dividir o processo em dois, controlando apenas o momento de escrita para garantir que os dois processos não escrevam juntos no arquivo de destino, tornando assim o arquivo comprimido irrecuperável.

No método de descompressão, após a leitura do primeiro bloco do arquivo comprimido, a função de descompressão é chamada e pode-se gerar um novo processo que irá ler e descomprimir o segundo bloco enquanto ocorre a descompressão do primeiro, já que não há comunicação entre nenhuma variável dos dois blocos. Dessa forma, seria necessário controlar apenas o reordenamento dos arquivos auxiliares para formar o arquivo original.

5.1. Execução e arquivos

A fim de possibilitar que em implementações e estudos futuros, seja possível realizar uma comparação com a forma como esse teste aconteceu, esta implementação foi realizada em C++, com compilador gcc 9.3.0 e os testes foram realizados em um terminal Ubuntu 20.04 LTS para Windows com um processador AMD Ryzen 7 3700U e 6 GB de RAM livre. A compressão para cada arquivo, em todos os métodos, foi realizada de forma separada, sem que houvesse compressões simultâneas, a fim de evitar disputa de recursos e consequente interferência no resultado final, principalmente na questão do tempo envolvido para o processamento.

Os arquivos para teste (IHNA, 2023) representam o DNA da espécie *Rattus norvegicus* e foram obtidos nos arquivos do ENA(European Nucleotide Archive), o que ajuda a garantir que os testes deste artigo foram realizados com arquivos reais e geneticamente relevantes.

6. Conclusão

Após a implementação do método proposto e análises de seu desempenho em comparação a outros métodos, tanto no arquivo completo, como em pontos-chave, pode-se atestar que o

resultado esperado foi obtido. Através do estudo do método LZW e do formato do arquivo FASTQ, seu pré-processamento e alteração do fluxo do método original, conseguiu-se gerar uma variante deste compressor, que apresenta resultados mais expressivos que os algoritmos de uso geral.

É importante lembrar ainda, que esse resultado, melhor do que os métodos-alvo, foi obtido utilizando-se um dicionário sem qualquer estratégia de substituição de valores, coisa que os métodos confrontados fazem. Dessa forma, o resultado torna-se ainda mais satisfatório, pois em um cenário de desigualdade, houve vantagem do método proposto.

Percebendo a vantagem do método proposto em relação aos métodos confrontados, quando se trata de eficácia de compressão, é interessante que futuras implementações foquem na redução do tempo de execução, sendo a paralelização, como explicado anteriormente, uma forma de obter melhor desempenho.

Como trabalhos futuros, pretende-se testar o algoritmo com diferentes tamanhos de dicionário, visto que, nos experimentos realizados, os índices do dicionário possuíam tamanho de 16 bits e a diminuição desse tamanho, embora implique em menos sequências no dicionário, diminui o tamanho dos índices que serão escritos no arquivo comprimido. Também é interessante analisar a divisão dos blocos, testando outras combinações ou utilizando outras técnicas para os blocos 1,3 e 4, como o próprio zip ou o rar.

Referências

ANWAR, Fahmi; FADLIL, Abdul; RIADI, Imam. Image Quality Analysis of PNG Images on WhatsApp Messenger Sending. Telematika., v. 14, n. 1, p. 1-12, 2021.

CÂMARA, Redes–Prof Marco. Criptografia E Compressão de Dados. 2009.

COCK, Peter JA et al. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic acids research, v. 38, n. 6, p. 1767-1771, 2010.

COLLINS, Francis S.; MORGAN, Michael; PATRINOS, Aristides. The Human Genome Project: lessons from large-scale biology. Science, v. 300, n. 5617, p. 286-290, 2003.

DEOROWICZ, Sebastian; GRABOWSKI, Szymon. Data compression for sequencing data. Algorithms for Molecular Biology, v. 8, p. 1-13, 2013.

DHEEMANTH, H. N. LZW data compression. American Journal of Engineering Research, v. 3, n. 2, p. 22-26, 2014.

ENES, D.; DOMINGUES, F.; ALÃO, T. M. Algoritmo de Huffman. Disponível em: <http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo -de-huffman/>. Acesso em: 14 maio. 2024.

EL ALLALI, Achraf; ARSHAD, Mariam. MZPAQ: a FASTQ data compression tool. Source code for biology and medicine, v. 14, p. 1-13, 2019.

FITRIYA, L. Anjar; PURBOYO, Tito Waluyo; PRASASTI, Anggunmeka Luhur. A review of data compression techniques. International Journal of Applied Engineering Research, v. 12, n. 19, p. 8956-8963, 2017.

IHNA. ENA Browser. www.ebi.ac.uk. Disponível em: <https://www.ebi.ac.uk/ena/browser/view/PRJNA904815>. Acesso em: 10 mar. 2024.

NISHAD, P. M.; CHEZIAN, R. Manicka. A vital approach to compress the size of DNA sequence using LZW (Lempel-Ziv-Welch) with fixed length binary code and tree structure. International Journal of Computer Applications, v. 43, n. 1, p. 7-9, 2012.

OSWAL, Savan; SINGH, Anjali ; KUMARI, Kirthi. Deflate Compression Algorithm. International Journal of Engineering Research and General Science, v. 4, n. 1, 2016.

SALOMON, David. Data Compression - The Complete Reference. Londres: Springer-Verlag, 2007.

ZEEH, Christina. The lempel ziv algorithm. In: Famous Algorithms: http://tuxtina.de/files/seminar/LempelZiv.pdf. 2003.