Relatório Trabalho Prático de AEDS 3

Luis Phillip, Fernanda Rodrigues Dias Mariano

27 de junho de 2024

Resumo

Este é o resumo do relatório, onde uma breve descrição do trabalho realizado é apresentada. Este resumo deve incluir o objetivo do trabalho, uma breve descrição da metodologia usada e os principais resultados obtidos.

Sumário

1 Introdução

Este relatório documenta os trabalhos práticos realizados na disciplina de Algoritmos e Estruturas de Dados 3. O projeto desenvolvido envolve a implementação de várias técnicas de compressão e estruturas de dados.

1.1 Estrutura do Diretório

A organização dos arquivos do projeto é a seguinte:

- **.dist:** Diretório contendo arquivos de distribuição.
- **.vscode:** Diretório contendo configurações do Visual Studio Code.
- **Compact:** Diretório contendo implementações de algoritmos de compressão.
  - **huffman:** Implementação do algoritmo de compressão Huffman.
    - `compressed_file.huff`
    - `decompressed_file.txt`
    - `huffman_tree.huff`
  - `Huffman.java`
  - **lzw:** Implementação do algoritmo de compressão LZW.
    - `songs.db`
    - `songs.lzw`
  - `LZW.java`
- **Files:** Diretório contendo arquivos de índices e estruturas de dados persistentes.
  - `Indices`
  - `btree.json`
  - `dirPrint.dat`
  - `Genero.dat`
  - `GeneroList.dat`
  - `IndiceInvertido.dat`
  - `printBtree.dat`
  - `ExemploBtree.txt`

- **Source:** Diretório contendo a base de dados usada.
  - **DataBase:** Código relacionado ao banco de dados.
    * `bucket.db`
    * `directorio.db`
    * `indices.db`
    * `songs.db`
    * `songs.lzw`
    * `songs.rsa`
  - `recuperacao`
- **Store:** Diretório contendo ferramentas de armazenamento.
  - `Btree.java`
  - `Diretorio.java`
  - `IndiceInvertido.java`
- **Structures:** Diretório contendo as classes de estrutura de dados.
  - `Indices.java`
  - `MetaData.java`
  - `MetaIndice.java`
  - `Musica.java`
- **Tools:** Ferramentas auxiliares para o projeto.
  - `Arq.java`
  - `Logs.java`
  - `MockData.java`
- **TP4:** Diretório contendo implementações específicas do Trabalho Prático 4.
  - `BoyerMoore.java`
  - `RSA.java`
- `InterfaceMenu.java`
- `arquivo_decrypted.txt`
- `arquivo_encrypted.txt`

- `Main.java`
- `teste.txt`

2 Desenvolvimento

Esta seção deve descrever detalhadamente os trabalhos práticos realizados (TP1, TP2, TP3, TP4). Cada TP pode ser descrito em subseções individuais. Inclua diagramas, códigos e explicações detalhadas.

2.1 Trabalho Prático 1

No Trabalho Prático 1, construímos um sistema de armazenamento de dados retirados do 'Spotify'. As classes desenvolvidas foram as seguintes:

- **InterfaceMenu.java:** Uma classe com uma GUI (Interface Gráfica do Usuário) para proporcionar uma melhor experiência entre o usuário e o banco de dados. Foi muito trabalhoso implementar e deixar a interface agradável visualmente.

- **Arq.java:** O coração do nosso sistema, responsável por controlar o CRUD (Create, Read, Update, Delete) no registro. Todas as funcionalidades de baixo nível estão nesta classe, como chamadas de criptografia, compactação, busca de padrão, criação de índices e pesquisa na B-tree ou hash. A classe **Arq** inclui métodos para iniciar o arquivo (`Iniciar`), procurar músicas por ID (`FindSongID`), deletar músicas (`DeleteSong`), atualizar músicas (`UpdateSong`), adicionar registros (`addRegistro`) e recuperar músicas deletadas (`Recover`). Ela também gerencia a leitura sequencial de registros e a criação de índices.

- **MetaData.java:** Decidimos terceirizar o processo de escrever registros no arquivo de texto e empacotá-los em uma classe específica. A MetaData possui os metadados de um registro lido ou prestes a ser escrito. Para facilitar a sincronização, tanto Arq quanto MetaData compartilham o acesso ao objeto `RandomAccessFile` do registro principal. Assim, quando Arq muda o cursor de leitura, MetaData está sempre acompanhando o processo.

- **Musica.java:** Nossa classe para representar a entidade música. Esta classe é vital na leitura e escrita, convertendo músicas para bytes e vice-versa.

- **Logs.java:** Uma classe destinada a emitir logs. Há muitos logs para serem emitidos, facilitando o debug. Implementamos outra interface para exibir logs positivos, alertas, logs de erro e detalhes menos importantes, como "Registro colocado no nó tal da B-tree".

- **MockData.java:** Classe destinada a ajudar no preenchimento de músicas, criando músicas como placeholders com nomes não tão aleatórios.

2.1.1 Descrição da Classe Arq

A classe **Arq** é a espinha dorsal do nosso sistema de armazenamento de dados. Ela é responsável por todas as operações de CRUD (Create, Read, Update, Delete) nos registros de músicas. Aqui estão os principais componentes e funcionalidades da classe:

- **Iniciar():** Inicializa o arquivo de armazenamento (`RandomAccessFile`) e configura a classe **MetaData**. Também inicializa vários componentes auxiliares como **Indices**, **IndiceInvertido**, **Diretorio**, **Btree**, **BoyerMoore**, e até mesmo algoritmos de criptografia e compactação como **Huffman**, **LZW**, e **RSA**.
- **FindSongID(int ID):** Procura uma música pelo seu ID, utilizando a classe interna **DataFinder**, que armazena os metadados e a música encontrada.
- **DeleteSong(int id):** Marca uma música como deletada (lápide) ao escrever um valor booleano verdadeiro no início do registro, indicando que o registro está deletado.
- **UpdateSong(int ID, String newSong):** Atualiza uma música existente. Se o novo registro for menor ou igual em tamanho ao registro existente, ele sobrescreve o antigo. Caso contrário, marca o antigo como deletado e adiciona o novo registro no final do arquivo.
- **addRegistro(String str):** Adiciona um novo registro de música ao final do arquivo, atualiza o último ID, cria índices e atualiza a B-tree, o hash extensível (Diretorio) e o índice invertido.
- **addRegistroExistenteEOF(Musica newSong):** Adiciona um registro existente ao final do arquivo, atualizando a B-tree e o hash extensível com a nova posição do registro.
- **Recover(int id):** Recupera uma música marcada como deletada, revertendo o valor da lápide para falso.
- **IniciarLeituraSequencial():** Posiciona o cursor do arquivo no início dos registros, pulando o último ID.

- **getNextRegistro():** Lê o próximo registro válido do arquivo, ignorando registros marcados como deletados.
- **writeLastID(int id):** Escreve o último ID no início do arquivo, garantindo que novos registros recebam IDs únicos e sequenciais.

A classe **Arq** também inclui métodos para ler todos os registros (`gettAllSongs`), pesquisar músicas utilizando a B-tree (`searchBtree`) e o hash extensível (`searchHash`), além de inserir todos os índices em buckets (`insertIntoBucket`) e adicionar gêneros aos índices invertidos (`addGenres`).

Esta classe é fundamental para o funcionamento eficiente do sistema de armazenamento de dados, garantindo a integridade e a rápida recuperação dos registros.

2.2 Trabalho Prático 2

O TP2 exigiu uma força mental e física que eu não sabia que tinha. Implementar a B-tree do zero, sem conhecimento prático prévio, foi um grande desafio. Vamos começar devagar.

Primeiro, desenvolvi a classe **MetaIndice**. Esta classe representa um índice de um registro, contendo sua posição e ID. Ela escreve no arquivo de acesso aleatório (`RandomAccessFile`, `raf`) fornecido como parâmetro. Passei 14 horas em um único dia tentando implementar a ordenação. Falhei inicialmente devido à falta de estudo adequado sobre o funcionamento da ordenação, o que acabou sendo muito trabalhoso. Consegui implementar a ordenação com uma perda de 2% dos registros. Nesse ponto, minha sanidade já estava comprometida, mas ainda precisava implementar a B-tree com a classe **Btree.java**.

Você já debugou uma B-tree? É um verdadeiro inferno. Enfrentei inúmeros bugs, mas não desisti. Na classe **Btree.java**, abstraí o máximo possível para lidar pouco com a escrita em arquivo e focar mais na lógica. Fiz o construtor da classe **Node** escrever o nó no arquivo e depois editar e reescrever. Assim, a alocação ficou muito mais fácil. Cada nó possui um atributo de localização no arquivo de texto. A árvore insere com 100% de precisão (sem perdas) e a pesquisa é extremamente eficiente. Com 32.000 registros, a árvore tem apenas 6 níveis de altura, o que é impressionante. Na remoção, fui menos esforçado e apenas marquei o registro como lápide no arquivo principal, deixando o arquivo no nó.

A implementação de **Diretorio.java** foi tranquila após a B-tree. Segui seus slides e a implementação foi bem simples. O diretório é um sistema de hash extensível que segue os buckets e profundidade local e global. A inserção também é 100% eficaz, e a remoção segue o mesmo princípio da B-tree.

Obs: Ambos os arquivos podem ser visualizados, pois fiz uma função para escrever a B-tree e o hash em arquivos de texto, localizados em `FILES/printbtree.info` e `FILES/dirPrint.dat`.

IndiceInvertido.java é uma classe para armazenar índices de músicas, usando seu gênero como atributo de agregação. Foi implementada utilizando uma lista ligada em arquivo de texto, onde o registro possui o índice daquele gênero e o ponteiro para o próximo registro. É muito lenta, então consegui inserir poucos registros, mostrando que uma lista ligada em arquivo de texto não compensa.

Indices.java é uma classe para coordenar os índices como um todo, pois **MetaIndice** representa um registro único e não foi feita para coordenar o arquivo inteiro. **Indices** faz isso com funções para pegar todos os índices, escrever índice de `String` (no caso do **IndiceInvertido** foi útil), buscar índice, etc.

2.2.1 Implementação da Classe Btree

A classe **Btree** foi desenvolvida para implementar uma árvore B, utilizada para gerenciar a indexação dos registros de forma eficiente. A árvore B foi configurada com ordem 8, significando que cada nó pode ter até 8 chaves. Os principais componentes da classe **Btree** incluem:

- **Node:** Representa um nó na árvore B. Cada nó contém um array de chaves (`MetaIndice`) e um array de ponteiros para os filhos. A classe **Node** inclui métodos para adicionar chaves em nós folha e não-folha, dividir nós quando estão cheios e escrever os nós no arquivo.
- **start():** Inicializa a árvore B, configurando a ordem, o arquivo de armazenamento (`RandomAccessFile`) e o nó raiz. Se o arquivo já contém dados, a raiz é lida do arquivo; caso contrário, uma nova raiz é criada.
- **add(MetaIndice metaIndice):** Adiciona um novo índice à árvore B, começando pela raiz e descendo pela árvore até encontrar o local apropriado para inserção.
- **search(int id):** Pesquisa por um índice na árvore B, começando pela raiz e descendo pelos nós até encontrar o índice ou determinar que ele não está presente.
- **updateIndex(MetaIndice metaIndice):** Atualiza a posição de um índice existente na árvore B.
- **printTree():** Imprime a estrutura da árvore B em um arquivo de texto para visualização e depuração.

2.2.2 Implementação da Classe Diretorio

A classe **Diretorio** foi desenvolvida para implementar um diretório de hash extensível, usado para gerenciar a indexação dos registros através de buckets. Os principais componentes da classe **Diretorio** incluem:

- **Bucket:** Representa um bucket no sistema de hash. Cada bucket contém chaves e posições de registro. A classe **Bucket** inclui métodos para adicionar e remover chaves, escrever e ler buckets do arquivo e resetar buckets.
- **start():** Inicializa o diretório, configurando a profundidade global e os arquivos de armazenamento (`RandomAccessFile`). Se não houver buckets existentes, novos buckets são criados.
- **inserir(MetaIndice reg):** Insere um novo índice no diretório, calculando o hash e determinando o bucket apropriado para a inserção. Se o bucket estiver cheio, ele é dividido.
- **searchID(int id):** Pesquisa por um índice no diretório, retornando verdadeiro se o índice for encontrado.
- **updateIndex(MetaIndice reg):** Atualiza a posição de um índice existente no diretório.
- **printDir():** Imprime a estrutura do diretório de hash em um arquivo de texto para visualização e depuração.
- **duplicarDiretorio():** Duplica a estrutura do diretório quando a profundidade global é atingida.

2.3 Trabalho Prático 3

No Trabalho Prático 3, foram implementados dois algoritmos de compressão: LZW e Huffman. A seguir, documentamos os principais métodos e classes usados nesses algoritmos.

2.3.1 Implementação do Algoritmo LZW

A classe **LZW** foi desenvolvida para realizar a compactação e descompactação de arquivos utilizando o algoritmo LZW (Lempel-Ziv-Welch). O processo de compactação começa com a leitura dos registros de um arquivo, pulando os

primeiros 4 bytes e armazenando todos os registros em uma lista. Esses registros são concatenados em um único array de bytes, com um delimitador nulo entre eles.

O método `compressBytes` é responsável por realizar a compressão dos dados, utilizando um dicionário para mapear sequências de caracteres a códigos inteiros. À medida que os dados são lidos, novas sequências são adicionadas ao dicionário, e os códigos correspondentes são armazenados em uma lista.

Finalmente, os dados compactados são escritos em um novo arquivo usando o método `writeCompressedFile`. O processo de descompactação, por outro lado, envolve a leitura dos dados compactados, a reconstrução do dicionário e a conversão dos códigos de volta para a forma original.

A classe também inclui métodos auxiliares, como `readAllRecords`, que lê todos os registros de um arquivo, e `decompress`, que gerencia o processo de descompactação.

2.3.2 Implementação do Algoritmo Huffman

A classe **Huffman** foi desenvolvida para realizar a compactação e descompactação de arquivos utilizando o algoritmo de Huffman. A compactação começa com a leitura dos registros de um arquivo, calculando as frequências dos bytes presentes nos dados. Essas frequências são usadas para construir a árvore de Huffman.

O método `buildHuffmanTree` constrói a árvore de Huffman a partir das frequências dos bytes, utilizando uma fila de prioridade para combinar os nós de menor frequência. A árvore resultante é usada para gerar códigos de Huffman únicos para cada byte.

Os códigos de Huffman são armazenados em um mapa (`huffmanNodes`), e os dados são compactados substituindo cada byte pelo seu código de Huffman correspondente. Os dados compactados são então escritos em um novo arquivo.

O método `decompress` realiza a descompactação, lendo a árvore de Huffman de um arquivo separado e utilizando-a para decodificar os dados compactados de volta à sua forma original. A árvore de Huffman é lida do arquivo usando o método `readHuffmanTree`, que reconstrói a árvore a partir do fluxo de entrada.

Para manipulação de bits durante a compactação e descompactação, foram utilizadas as classes **BitOutputStream** e **BitInputStream**, que gerenciam a escrita e leitura de bits individuais em um fluxo de bytes.

- **LZW.java:** - `start`: Inicia a compactação dos registros de um arquivo. - `readAllRecords`: Lê todos os registros do arquivo. - `compressBytes`: Realiza a compressão LZW de um array de bytes. - `writeCompressedFile`:

Escreve os dados compactados em um novo arquivo. - `decompress`: Descompacta os dados do arquivo compactado.

- **Huffman.java:** - `start`: Inicia a compactação dos registros de um arquivo. - `readByteFrequencies`: Lê as frequências dos bytes no arquivo. - `buildHuffmanTree`: Constrói a árvore de Huffman a partir das frequências dos bytes. - `generateHuffmanCodes`: Gera os códigos de Huffman a partir da árvore. - `compressFile`: Compacta o arquivo utilizando os códigos de Huffman. - `writeHuffmanTreeToFile`: Escreve a árvore de Huffman em um arquivo separado. - `writeCompressedFile`: Escreve os dados compactados em um novo arquivo. - `decompress`: Descompacta os dados do arquivo compactado. - `readHuffmanTree`: Lê a árvore de Huffman do arquivo.

- **HuffmanNode.java:** - Classe que representa um nó na árvore de Huffman, contendo o byte, sua frequência e seus filhos esquerdo e direito.

- **BitOutputStream.java:** - Classe para manipulação de bits em um fluxo de saída.

- **BitInputStream.java:** - Classe para manipulação de bits em um fluxo de entrada.

Esta implementação dos algoritmos de compressão LZW e Huffman permitiu uma eficiente redução do tamanho dos arquivos, mantendo a integridade dos dados durante os processos de compactação e descompactação.

2.4 Trabalho Prático 4

No Trabalho Prático 4, implementamos dois componentes principais: a busca de padrões utilizando o algoritmo de Boyer-Moore e a criptografia RSA. A seguir, detalhamos a implementação desses componentes.

2.4.1 Implementação do Algoritmo de Boyer-Moore

A classe **BoyerMoore** foi desenvolvida para realizar a busca de padrões em textos, utilizando o algoritmo de Boyer-Moore. Este algoritmo é conhecido por sua eficiência em buscas, especialmente em textos grandes. A classe inclui os seguintes componentes e funcionalidades:

- **Start():** Inicializa o array local com todas as músicas disponíveis no sistema, utilizando o método `Arq.gettAllSongs()`.
- **FindPattern(String pat):** Inicia a busca de um padrão especificado (`pat`) nas músicas, chamando métodos para pré-processamento do padrão e para a busca propriamente dita.

- **preprocessPattern(String pat):** Preprocessa o padrão fornecido para criar os arrays `badChar` e `goodSuffix`, necessários para a execução eficiente do algoritmo de Boyer-Moore.
- **preprocessBadChar(String pat):** Cria o array `badChar`, que armazena as posições das ocorrências dos caracteres no padrão, ajudando a determinar os deslocamentos quando um caractere não correspondente é encontrado.
- **preprocessGoodSuffix(String pat):** Cria o array `goodSuffix`, que armazena informações sobre os sufixos bons no padrão, ajudando a determinar os deslocamentos quando ocorre um sufixo correspondente.
- **searchPatternInSongs():** Executa a busca do padrão em todas as músicas disponíveis, verificando tanto o nome quanto o artista das músicas.
- **searchInText(String text):** Realiza a busca do padrão em um texto específico, retornando verdadeiro se o padrão for encontrado.

Esta implementação permite uma busca eficiente de padrões em grandes volumes de texto, como os nomes e artistas das músicas no sistema.

2.4.2 Implementação da Criptografia RSA

A classe **RSA** foi desenvolvida para realizar a criptografia e descriptografia de arquivos utilizando o algoritmo RSA, que é um método de criptografia de chave pública amplamente utilizado. A classe inclui os seguintes componentes e funcionalidades:

- **start():** Inicializa as chaves pública e privada necessárias para a criptografia e descriptografia. As chaves são geradas utilizando dois grandes números primos (`p` e `q`), e a chave pública (`e`) e a chave privada (`d`) são calculadas com base em `p` e `q`.
- **encrypt(String inputFile, String outputFile):** Criptografa um arquivo de entrada e salva o resultado criptografado em um arquivo de saída. O processo de criptografia é realizado em blocos para garantir a segurança dos dados.
- **decrypt(String inputFile, String outputFile):** Descriptografa um arquivo de entrada criptografado e salva o resultado descriptografado em um arquivo de saída. O processo de descriptografia também é realizado em blocos.

- **processFile(String inputFile, String outputFile, boolean encrypt):** Método auxiliar para criptografar ou descriptografar um arquivo. Ele lê os dados do arquivo de entrada em blocos, aplica a operação de criptografia ou descriptografia e escreve o resultado no arquivo de saída.

- **getPhi(BigInteger p, BigInteger q):** Calcula a função totiente de Euler (`phi`) para os números primos `p` e `q`.

- **largePrime(int bits):** Gera um grande número primo com o número especificado de bits.

- **gcd(BigInteger a, BigInteger b):** Calcula o maior divisor comum (GCD) de dois números.

- **extEuclid(BigInteger a, BigInteger b):** Executa o algoritmo de Euclides estendido para encontrar o GCD e os coeficientes de Bézout de `a` e `b`.

- **genE(BigInteger phi):** Gera a chave pública `e`, que é coprima com `phi`.

A implementação da criptografia RSA garante a segurança dos dados, permitindo a proteção e a integridade dos arquivos no sistema.

Conclusão

No Trabalho Prático 4, implementamos funcionalidades avançadas de busca de padrões e criptografia. A classe **BoyerMoore** melhora significativamente a eficiência das buscas no sistema, enquanto a classe **RSA** assegura a segurança dos dados armazenados. Essas implementações são essenciais para a robustez e a confiabilidade do sistema desenvolvido.

3 Testes e Resultados

Nesta seção, serão descritos os testes realizados no sistema desenvolvido e os resultados obtidos. Os testes incluem a comparação de tempos de busca em B-tree e Hash, bem como a análise das taxas de compressão e uso de memória utilizando os algoritmos Huffman e LZW. Gráficos e tabelas são apresentados para validar a implementação.

3.1 Teste de Pesquisa: B-tree vs Hash

Foram realizados testes de busca utilizando B-tree e Hash. Os tempos de execução para cada teste foram medidos e comparados. Os resultados são apresentados na Tabela 1 e no Gráfico 1.

Tabela 1: Tempos de Busca em B-tree e Hash

Teste	**Tempo de Busca em B-tree (ms)**	**Tempo de Busca em Hash (ms)**
1	929	1077
2	830	1100
3	840	1072
4	839	1070
5	836	1075
6	859	1127
7	861	1095
8	874	1094
9	843	1091
10	843	1100

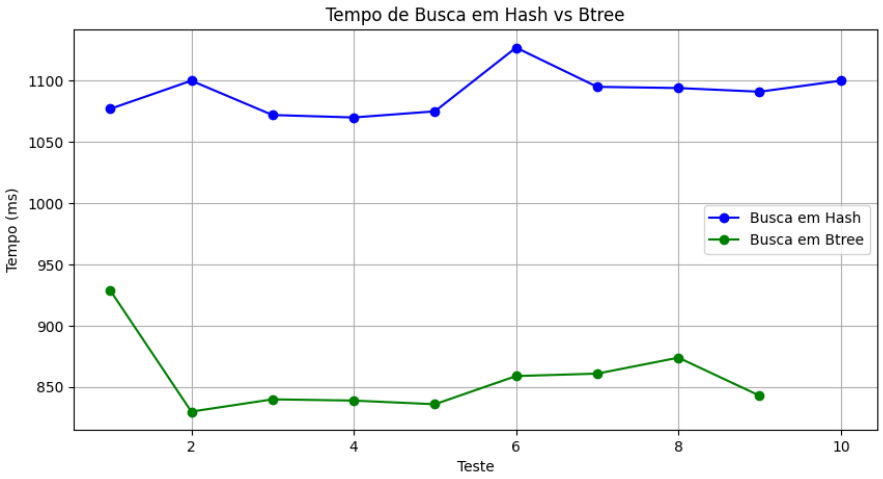

Figura 1: Comparação dos tempos de busca em B-tree e Hash

Os resultados mostram que a busca utilizando B-tree é consistentemente mais rápida em comparação com a busca utilizando Hash.

3.2 Teste de Compressão: Huffman vs LZW

Foram realizados testes de compressão utilizando os algoritmos Huffman e LZW. As taxas de compressão foram medidas para cada algoritmo. Os resultados são apresentados na Tabela 2 e no Gráfico 2.

Tabela 2: Taxas de Compressão: Huffman vs LZW

Algoritmo	**Taxa de Compressão (%)**
Huffman	74.58
LZW	89.74

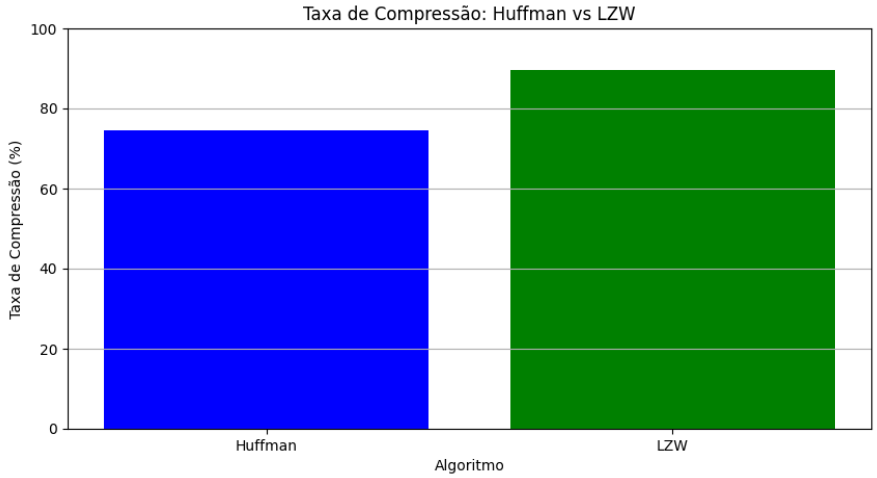

Figura 2: Comparação das taxas de compressão entre Huffman e LZW

Os resultados indicam que o algoritmo LZW apresenta uma taxa de compressão mais alta em comparação com o algoritmo Huffman, tornando-o mais eficiente na redução do tamanho do arquivo.

3.3 Uso de Memória e Velocidade de Compressão

Além das taxas de compressão, foram medidos o uso de memória e os tempos de execução para os algoritmos Huffman e LZW. Os resultados são apresentados na Tabela 3 e nos Gráficos 3 e **??**.

Os resultados indicam que o algoritmo Huffman é mais eficiente em termos de uso de memória e tempo de execução em comparação com o algoritmo LZW, apesar deste último apresentar uma taxa de compressão mais alta.

Tabela 3: Uso de Memória e Tempo de Execução: Huffman vs LZW

Algoritmo	**Uso de Memória (MB)**	**Tempo de Execução (ms)**
Huffman	79	4284.43
LZW	203	10395.66
Huffman	13	4317.61
LZW	234	10231.23
Huffman	73	4258.85
LZW	213	10291.99
Huffman	71	4287.57
LZW	241	10184.59
Huffman	43	4290.98
LZW	215	10224.28

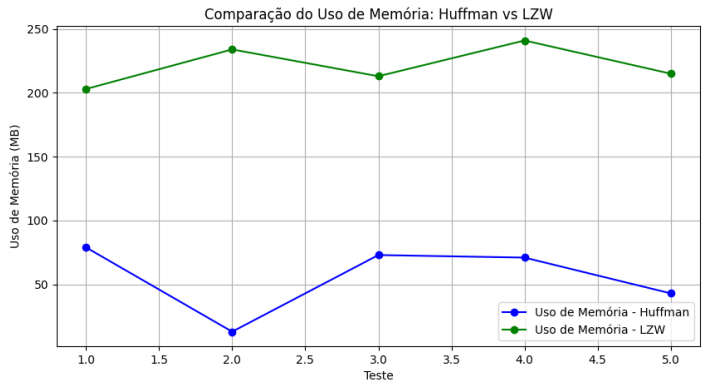

Figura 3: Comparação do uso de memória entre Huffman e LZW

3.4 Análise dos Resultados

A análise dos resultados revela que a busca utilizando B-tree não só é mais rápida, mas também mais consistente em comparação com Hash. Em termos de compressão, o algoritmo LZW mostrou-se mais eficiente em reduzir o tamanho do arquivo, embora consuma mais memória durante o processo de compressão e tenha um tempo de execução maior. Os gráficos fornecem uma visualização clara das diferenças de desempenho entre os algoritmos testados.

4 Referências

Não usei quase nenhuma a não ser os slides do grande professor Hayala e vídeos do professor Kutova. também a documentação java para mexer no

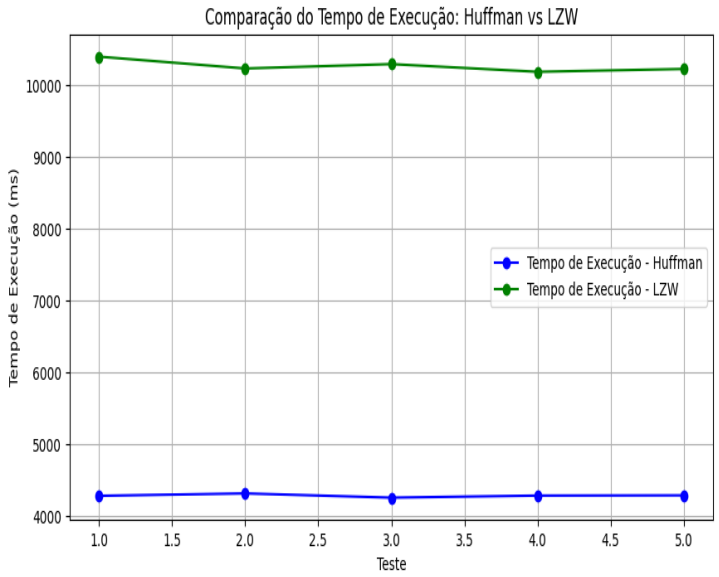

Figura 4: Comparação do tempo de execução entre Huffman LZW

GUI.