Relatório – Estrutura de Dados II

Projeto Nº02 – Verificador de Similaridade de Textos

Este relatório foi escrito para o projeto que compõe a N2 da disciplina Estrutura de Dados II do curso de Ciência da Computação Mackenzie (FCI).

GitHub: https://github.com/M4t3u5534/Verificador-de-Similaridade-de-Textos/tree/main

Aluno: Mateus Cerqueira Ribeiro - 10443901
Aluno: Pedro Henrique Carvalho Pereira - 10418861
Aluno: Gabriel Mires Camargo - 10436741
Turma: 04D - CC

1. INTRODUÇÃO

Neste projeto, foi desenvolvido um sistema capaz de identificar o grau de similaridade entre documentos de texto, aplicando técnicas de pré-processamento dos dados textuais (como normalização, tokenização e remoção de *stop-words*) e o cálculo da porcentagem de similaridade por meio do método do Cosseno. Cada documento possui seu próprio vocabulário, que é armazenado em uma tabela *hash* implementada manualmente, utilizando a técnica de *hash* duplo para lidar com colisões e melhorar a dispersão das chaves.

Com os dados organizados e a similaridade calculada, os resultados obtidos na comparação entre dois ou mais textos são inseridos em uma árvore *AVL*, que mantém, em cada nó, os pares de documentos comparados ordenados pelo grau de similaridade. Ao final, apenas a última execução é salva em um arquivo .txt (*log*), que contém um resumo do comando passado pelo usuário.

O objetivo principal desse projeto é comparar, organizar os documentos trabalhando os resultados em diferentes estruturas de dados. Toda a solução foi estruturada de forma modular, dividida em sete classes que serão detalhadas ao longo do relatório: *AVL.java, ComparadorDeDocumentos.java, Documento.java, Main.java, Node.java, Resultado.java* e *TabelaHash.java*.

2. METODOLOGIA

Nesta seção, serão desenvolvidos e detalhados o pré-processamento de dados, as estruturas e entradas de dados e o método de comparação para cálculo da similaridade entre documentos.

2.1 Descrição da Tabela Hash

A tabela *hash*, em seu princípio, tem objetivo armazenar as palavras de forma eficiente, possibilitando uma busca rápida durante qualquer processo necessário. No nosso projeto, ela foi implementada com o método de *hashing* duplo para controle de dispersão e tratamento de colisões, pois garante que cada palavra seja mapeada mesmo quando há chaves semelhantes ou nos casos de agrupamento.

Para o cálculo da posição final na tabela, deve-se passar por 3 partes (ou 3 funções), sendo elas: h1; h2; *hash*.

A primeira parte (h1) é a função *h1(String chave)* que aplica o *hash* multiplicando, a cada laço do *for*, o valor acumulado (hash - inicializado em 0) por 31 e somando o código ASCII do caractere atual.

```
// Primeira fucao hash
private int h1(String chave) {
    int hash = 0;
    for (char c : chave.toCharArray()) hash = (31 * hash + c) % m; // hash clássico
    return hash;
}
```

Para complementar essa dispersão e eliminar uma gama maior de colisões, a implementação do método utiliza uma segunda função *h2(String chave)*. Essa segunda função usa um outro multiplicador (37) e garante que o retorno de h2 não prenda chaves à mesma posição.

```
// Segunda funcao hash
private int h2(String chave) {
    int hash = 0;
    for (char c : chave.toCharArray()) hash = (37 * hash + c) % (m - 1); // menor que m
    return hash + 1; // garante que nunca seja zero
}
```

Agora, na terceira parte, acontece a junção dessas duas funções, enfim calculando a posição final. O cálculo da posição final é feito pela função *hash(String chave, int i)* que combina os resultados da seguinte forma:

```
// Função de hash dupla
private int hash(String chave, int i) {
    return (h1(chave) + i * h2(chave)) % m;
}
```

Prosseguindo, temos o processo de inserção concorda com a lógica de direcionamento e a complementa. A inserção realiza uma série de verificações antes de adicionar as entradas na classe interna *Entry*, como pares de chave e valor, onde o valor significa a quantidade de aparições da palavra no documento.

```java
public class TabelaHash {
    private Entry[] tabela;
    private int m; // tamanho da tabela
    private int n; // numero de elementos
    private int colisoes = 0; // numero de colisoes

    // Contrutor
    public TabelaHash(int tamanho) {
        this.m = tamanho;
        this.tabela = new Entry[m];
        this.n = 0;
    }
```

```java
    public static class Entry {
        String chave;
        int valor;

        Entry(String chave) {
            this.chave = chave;
            this.valor = 1;
        }
    }
}
```

Para o caso de uma mesma palavra aparecer em outra parte no documento, a comparação *tabela[posicao].chave.equals(chave)* capta essa igualdade e apenas incrementa o contador para aquela palavra.

```java
// Inserir elemento
public void inserir(String chave, int i) {
    if (n >= m) {
        System.out.println(x: "Tabela cheia!");
        return;
    }

    int posicao = hash(chave, i);

    if (tabela[posicao] == null) {
        tabela[posicao] = new Entry(chave);
        n++;
        return;
    }

    if (tabela[posicao].chave.equals(chave)) {
        tabela[posicao].valor++;
        return;
    }

    colisoes++;
    inserir(chave, i + 1);
}
```

No caso de uma colisão ocorrer, o contador de colisões *colisoes* é incrementado e é feita uma segunda tentativa, de forma recursiva, para inserir a chave na tabela, sempre somando +1 à variável *i*, até encontrar uma posição livre.

Essa lógica de incremento aparece novamente na busca de uma chave no *for* implementado.

```java
// --- Busca usando double hashing ---
public int buscar(String chave) {
    for (int i = 0; i < m; i++) {
        int pos = hash(chave, i);

        if (tabela[pos] == null) {
            return -1;
        }

        if (tabela[pos].chave.equals(chave)) {
            return tabela[pos].valor;
        }
    }
    return -1;
}
```

2.2 Descrição do Pré-Processamento

Agora que foi detalhado o armazenamento das palavras, é imprescindível compreender o pré-processamento que é feito na classe *Documento.java*. Nesse pré-processamento, o arquivo bruto é transformado em um conjunto limpo de *tokens* (palavras - chaves - vocabulário) que, então, serão inseridos na tabela *hash*.

Assim que um objeto Documento é criado, o caminho do arquivo é armazenado, o arquivo é lido linha por linha e então armazenado em um único texto contínuo.

```java
public Documento(Path caminho) throws IOException {
    this.caminhoArquivo = caminho;
    this.nomeArquivo = caminho.getFileName().toString();

    // le as linhas do arquivo
    List<String> linhas = Files.readAllLines(caminho);

    // junta as linhas em um texto unico
    String textoCompleto = String.join(delimiter: " ", linhas);
```

Em seguida, inicia-se a formatação do texto (ainda no construtor):

```java
    // normaliza o texto
    String textoNormalizado = normalizarTexto(textoCompleto);

    // tokeniza divide em palavras
    String[] palavras = textoNormalizado.split(regex: "\\s+");
```

O método *normalizarTexto()* é chamado para padronizar as palavras. E com o texto normalizado retornado é feita a tokenização usando os espaços entre palavras como separador. Abaixo, o método de normalização de texto:

```
private String normalizarTexto(String texto) {
    // converte para minusculas
    texto = texto.toLowerCase();

    // retira tudo que nao estiver entre a-z e 0-9
    texto = texto.replaceAll(regex: "[^a-záàâãéèêíóôõúç0-9\\s]", replacement: "");

    // remove espacos extras (multiplos espacos viram um)
    texto = texto.trim().replaceAll(regex: "\\s+", replacement: " ");

    return texto;
}
```

Agora, com as palavras separadas e armazenadas individualmente, é criada a tabela *hash* (no construtor - depois da tokenização) é feita a inserção através do método *processarPalavras(palavras)*:

```
    // cria a tabela hash com tamanho apropriado
    // multiplicamos por 2 para manter fator de carga baixo
    int tamanhoTabela = Math.max(palavras.length * 2, b: 10); // minimo de 10
    this.vocabulario = new TabelaHash(tamanhoTabela);

    // processa e insere as palavras na tabela hash
    processarPalavras(palavras);
}
```

A inserção é feita percorrendo o array e inserindo cada palavra:

```
// contabiliza a frequencia de cada palavra
// palavras representa o array de palavras a serem processadas
private void processarPalavras(String[] palavras) {
    for (String palavra : palavras) {
        // pula palavras vazias
        if (palavra.isEmpty()) {
            continue;
        }

        // pula palavras de parada
        if (STOP_WORDS.contains(palavra)) {
            continue;
        }

        // insere na tabela hash cria se nao existir
        vocabulario.inserir(palavra, i: 0);
    }
}
```

O método de inserção na tabela *hash* e direcionamento dos dados já foi discutido e detalhado no tópico anterior.

2.3 Descrição da Árvore AVL

A classe *AVL.java* foi implementada para armazenar os pares de documentos de acordo com o valor de similaridade calculado na classe *ComparadorDeDocumentos.java*. Cada nó (Node) dessa árvore armazena um valor de similaridade, uma lista de objetos Resultado e referências (*left*, *right*) que serão apontadas para os filhos esquerdo e direito, respectivamente, do nó atual. Sempre que um novo *Resultado* for inserido a árvore segue balanceada.

A seguir, a estrutura do objeto *Resultado*:

```java
public class Resultado {
    private String doc1;
    private String doc2;
    private double similaridade;

    public Resultado(String doc1, String doc2, double similaridade) {
        this.doc1 = doc1;
        this.doc2 = doc2;
        this.similaridade = similaridade;
    }

    public double getSimilaridade() { return similaridade; }
    public String getDoc1() { return doc1; }
    public String getDoc2() { return doc2; }

    @Override
    public String toString() {
        return doc1 + " <-> " + doc2 + " = " + String.format(format: "%.2f", similaridade);
    }
}
```

A inserção na classe *AVL.java* é feita de forma recursiva através de dois métodos:

```java
// inserção
public void insert(Resultado res) {
    raiz = insereRec(raiz, res);
}

private Node insereRec(Node node, Resultado res) {
    if (node == null) return new Node(res);

    if (res.getSimilaridade() < node.similaridade) {
        node.left = insereRec(node.left, res);

    } else if (res.getSimilaridade() > node.similaridade) {
        node.right = insereRec(node.right, res);

    } else {node.addResultado(res);}


    node.atualiza_h();

    // balanceamento
    int balanco = node.fator_balanceamento(); // dif entre alturas

    if (balanco > 1 && res.getSimilaridade() < node.left.similaridade) {
        rotacoes_simples++;
        return node.rotacao_direita();
    } if (balanco < -1 && res.getSimilaridade() > node.right.similaridade) {
        rotacoes_simples++;
        return node.rotacao_esquerda();
    } if (balanco > 1 && res.getSimilaridade() > node.left.similaridade) {
        rotacoes_duplas++;
        node.left = node.left.rotacao_esquerda();
        return node.rotacao_direita();
    } if (balanco < -1 && res.getSimilaridade() < node.right.similaridade) {
        rotacoes_duplas++;
        node.right = node.right.rotacao_direita();
        return node.rotacao_esquerda();
    } return node;
}
```

A inserção é feita através da similaridade. Se a similaridade do novo *resultado* for menor que a do nó atual, a inserção segue para o filho esquerdo (*left*), caso contrário, segue para o filho à direita (*right*) e se a similaridade for igual ela é adicionada à lista de resultados do nó que já existe.

Após a inserção é necessário balancear a árvore. A altura é atualizada e o fator de balanceamento é calculado pela diferença entre as alturas do filho esquerdo e direito através da função *fator_balanceamento()* da classe *Node.java*. Então, se o fator estiver fora do intervalo [-1,1], deve-se rotacionar a árvore.

Para rotação simples à direita (quando um nó pesa à esquerda e o novo valor é menor que o valor do filho esquerdo) temos o trecho:

```
if (balanco > 1 && res.getSimilaridade() < node.left.similaridade) {
    rotacoes_simples++;
    return node.rotacao_direita();
```

Para rotação simples à esquerda (se o desbalanceamento é à direita e o novo valor é maior que o filho direito) temos o trecho:

```
} if (balanco < -1 && res.getSimilaridade() > node.right.similaridade) {
    rotacoes_simples++;
    return node.rotacao_esquerda();
```

Um caso de rotação dupla é necessário quando o nó que pesa à esquerda teve inserção na subárvore direita ou o contrário, o nó que pesa à direita teve inserção na subárvore esquerda.

Para rotação dupla à direita deve-se fazer uma rotação esquerda no filho esquerdo e depois uma rotação direita no nó desbalanceado. Temos o trecho:

```
} if (balanco > 1 && res.getSimilaridade() > node.left.similaridade) {
    rotacoes_duplas++;
    node.left = node.left.rotacao_esquerda();
    return node.rotacao_direita();
```

Para rotação dupla à esquerda, fazemos o oposto. Temos o trecho:

```
} if (balanco < -1 && res.getSimilaridade() < node.right.similaridade) {
    rotacoes_duplas++;
    node.right = node.right.rotacao_direita();
    return node.rotacao_esquerda();
```

Para cada seção de balanceamento, um contador do tipo da rotação foi implementado para controle e análise experimental.

As funções *rotacao_direita()* e *rotacao_esquerda()* foram implementadas no *Node*:

```
public Node rotacao_esquerda() {
    Node y = right;
    Node T2 = y.left;
    y.left = this;
    right = T2;
    atualiza_h();
    y.atualiza_h();
    return y;
}
```

```
44  public Node rotacao_direita() {
45      Node x = left;
46      Node T2 = x.right;
47      x.right = this;
48      left = T2;
49      atualiza_h();
50      x.atualiza_h();
51      return x;
52  }
```

A rotação direita é usada quando a subárvore esquerda está mais alta que a direita e a inserção foi feita na esquerda do filho esquerdo. Já a rotação esquerda é usada quando a subárvore direita está mais alta que a esquerda e a inserção foi feita na direita do filho direito.

2.4 Justificativa da métrica de similaridade escolhida

O método escolhido foi o Cosseno. Ele foi escolhido pela sua facilidade e implementação.

O cálculo da similaridade segue um algoritmo. Primeiro, devemos entender a estrutura em que os dados são armazenados. Cada documento é representado em um vetor composto pelas frequências das palavras. Então, a similaridade é calculada através do ângulo entre esses vetores.

No código desenvolvido, a primeira etapa é calcular o produto escalar entre os vetores de frequência dos documentos.

```
// percorre todas as palavras do doc1
for (TabelaHash.Entry e1 : d1.getVocabulario().getTabela()) {
    if (e1 == null) continue; // ignora posições vazias da tabela hash

    // busca a frequência da mesma palavra no doc2
    int freq2 = d2.getVocabulario().buscar(e1.chave);
    if (freq2 < 0) freq2 = 0; // se a palavra não existir, considera frequência 0

    // atualiza o produto escalar (quant * freq)
    somaProduto += e1.valor * freq2;

    // atualiza a soma dos quadrados do doc1
    soma1 += e1.valor * e1.valor;
}
```

O *for* acima percorre as palavras do primeiro documento, recupera a frequência da mesma palavra no segundo e acumula o produto. Em seguida, calcula-se a magnitude (também chamada de norma) do vetor do primeiro documento através da soma dos quadrados das frequências (soma1).

Depois, no segundo *for*, calcula-se a magnitude do segundo documento:

```
// percorre todas as entradas do doc2 para calcular a soma dos quadrados
for (TabelaHash.Entry e2 : d2.getVocabulario().getTabela()) {
    if (e2 == null) continue; // posições vazias
    soma2 += e2.valor * e2.valor; // soma os quadrados das frequências
}
```

Por fim, a similaridade final é calculada dividindo o produto escalar pelo produto das magnitudes dos vetores:

```
// Divide o produto escalar pelo produto das magnitudes (raiz quadrada das somas dos quadrados)
// 1e-10 é adicionado para evitar divisão por zero caso algum vetor seja nulo
return somaProduto / (Math.sqrt(soma1) * Math.sqrt(soma2) + 1e-10);
```

Este método utilizou a fórmula matemática abaixo para base de implementação do código:

$$similaridade(A,B) = \frac{A \,.\, B}{|A|\,|B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

No código:

-> *somaProduto* corresponde à somatória $\sum_{i=1}^{n} A_i B_i$

-> *soma1* e *soma2* correspondem às somas $\sum_{i=1}^{n} A_i^2$ e $\sum_{i=1}^{n} B_i^2$

-> *return* corresponde ao cálculo completo $\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$

3. DESCRIÇÃO DO ALGORITMO

Para explicar o algoritmo será feita uma descrição dos processos na *Main.java*, detalhando brevemente o fluxo de dados.

O algoritmo começa com a inicialização e leitura de parâmetros. O programa começa verificando se os argumentos necessários foram passados:

```
if (args.length < 3) {
    System.err.println("Argumentos faltantes: " + (3 - args.length));
    System.err.println(x: "Uso: java Main <diretorio_documentos> <limiar> <modo> [argumentos_opcionais]");
    return;
}
```

A entrada esperada pelo usuário se baseia na estrutura de escrita abaixo:

java Main <diretorio_documentos> <limiar> <modo> [argumentos_opcionais]

Onde:

- diretorio_documentos: pasta onde estão os arquivos a serem comparados.
- limiar: valor mínimo de similaridade para considerar um par relevante.
- modo: "topk" (retorna os K pares mais similares) ou "busca" (compara dois arquivos específicos).
- Argumentos opcionais: K para topk ou nomes de arquivos para busca.

Depois disso, são feitas as inicializações das variáveis com as informações passadas pelo usuário, verificação de existência do diretório e identificação do modo selecionado.

```java
// principais inicializações
String diretorio = args[0];
double limiar = Double.parseDouble(args[1]);
String modo = args[2].toLowerCase();
int topK = -1;
List<String> buscaArquivos = new ArrayList<>();

// SETUP
// topk
if (modo.equals(anObject: "topk")) {
    if (args.length < 4) {
        System.err.println(x: "Modo topK requer argumento K: java Main <diretorio> <limiar> topK <K>");
        return;
    }
    topK = Integer.parseInt(args[3]);

// busca
} else if (modo.equals(anObject: "busca")) {
    if (args.length < 5) {
        System.err.println(x: "Modo busca requer dois arquivos: java Main <diretorio> <limiar> busca <arquivo1> <arquivo2>");
        return;
    }

    buscaArquivos.add(args[3]);
    buscaArquivos.add(args[4]);
}

// verificações de diretório
Path dirPath = Paths.get(diretorio);
if (!Files.exists(dirPath) || !Files.isDirectory(dirPath)) {
    System.err.println("Diretório " + diretorio + " não encontrado.");
    return;
}
```

Em seguida inicia-se o processo de normalização com a leitura e representação dos documentos. Cada arquivo do diretório é lido e transformado em um objeto Documento. Dentro da classe *Documento.java*, é criada a tabela *hash* para armazenar o vocabulário normalizado e tokenizado.

```java
// carrega documentos
List<Documento> documentos = new ArrayList<>();
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dirPath)) {
    for (Path arquivo : stream) {
        if (Files.isRegularFile(arquivo)) {

            Documento doc = new Documento(arquivo);
            documentos.add(doc);

            // coleta dados de hash
            TabelaHash tabela = doc.getVocabulario();
            totalColisoesHash += tabela.getColisoes();
            fatoresCarga.add((double)tabela.getColisoes() / tabela.getTabela().length);

        }
    }System.out.println(x: "");
}
```

Ao armazenar todos os vocabulários de cada documento, é feito o cálculo de similaridade entre pares de documentos, usando a similaridade do cosseno para a medição, e em seguida cada similaridade por par de comparação é armazenado em uma estrutura Resultado a qual é inserida como nó em uma árvore *AVL*.

```java
// processamento de todos os pares
for (int i = 0; i < documentos.size(); i++) {
    for (int j = i + 1; j < documentos.size(); j++) {
        Documento d1 = documentos.get(i);
        Documento d2 = documentos.get(j);

        if (modo.equals(anObject: "busca")) {
            if (!((d1.getNomeArquivo().equals(buscaArquivos.get(index: 0)) && d2.getNomeArquivo().equals(buscaArquivos.get(index: 1))) ||
                  (d1.getNomeArquivo().equals(buscaArquivos.get(index: 1)) && d2.getNomeArquivo().equals(buscaArquivos.get(index: 0))))) {
                continue;
            }
        }

        // tempo da similaridade
        long t0 = System.nanoTime();
        double sim = ComparadorDeDocumentos.calcularSimilaridadeCosseno(d1, d2);
        long t1 = System.nanoTime();
        tempoTotalSimilaridade += (t1 - t0);

        if (sim >= limiar) {

            // medir inserção na AVL
            long tA1 = System.nanoTime();
            avl.insert(new Resultado(d1.getNomeArquivo(), d2.getNomeArquivo(), sim));
            long tA2 = System.nanoTime();
            tempoTotalInsercaoAVL += (tA2 - tA1);

            totalComparados++;
        }
    }
}
```

Por fim, é feita a coleta e ordenação final dos resultados e geração de arquivo de saída.

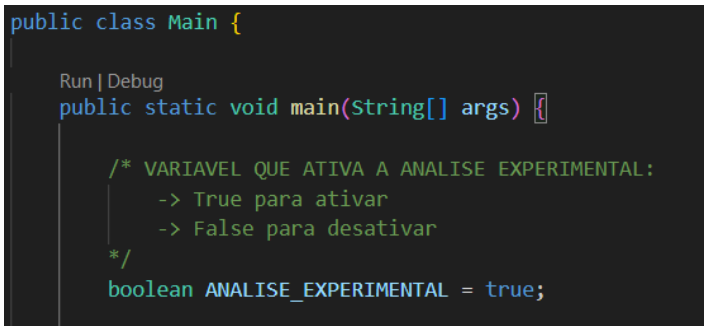
```java
// Coletar resultados em ordem decrescente
List<Resultado> resultados = new ArrayList<>();
coletarResultadosEmOrdem(avl.raiz, resultados);
resultados.sort((r1, r2) -> Double.compare(r2.getSimilaridade(), r1.getSimilaridade()));

if (modo.equals(anObject: "topk") && topK > 0 && resultados.size() > topK) {
    resultados = resultados.subList(fromIndex: 0, topK);
}

// Gerar arquivo resultado.txt em UTF-8 e imprimir no console
try (BufferedWriter writer = new BufferedWriter(
        new OutputStreamWriter(new FileOutputStream(name: "resultado.txt"), charsetName: "UTF-8"))) {
    writer.write(str: "=== VERIFICADOR DE SIMILARIDADE DE TEXTOS ===\n");
    writer.write("Total de documentos processados: " + documentos.size() + "\n");
    writer.write("Total de pares comparados: " + totalComparados + "\n");
    writer.write(str: "Função hash utilizada: hashMultiplicativo\n");
    writer.write(str: "Métrica de similaridade: Cosseno\n");
    writer.write("Pares com similaridade >= " + limiar + ":\n");
    writer.write(str: "--------------------------------\n");

    for (Resultado r : resultados) {
        writer.write(r.toString() + "\n");
        System.out.println(r);
    }
}

System.out.println(x: "\nResultados salvos em resultado.txt");
System.out.println("Rotações simples: " + avl.rotacoes_simples);
System.out.println("Rotações duplas: " + avl.rotacoes_duplas);
```

No caso do usuário escolher fazer a análise experimental, informações de tempo e quantidade de passos são coletadas no meio da *Main* e no final é feita a interpretação desses dados coletados.

```
if (ANALISE_EXPERIMENTAL) {

    long fimPrograma = System.currentTimeMillis();
    double tempoTotal = fimPrograma - inicioPrograma;

    System.out.println(x: "\n=== ANÁLISE EXPERIMENTAL ===");

    System.out.println("Tempo total do programa: " + tempoTotal + " ms");
    System.out.println("Tempo médio por comparação: " +
        (tempoTotalSimilaridade / 1_000_000.0) / totalComparados + " ms");

    System.out.println("Tempo total em similaridade: " +
        (tempoTotalSimilaridade / 1_000_000.0) + " ms");

    System.out.println("Tempo médio de inserção na AVL: " +
        (tempoTotalInsercaoAVL / 1_000_000.0) + " ms");

    System.out.println("Total de colisões (todos os documentos): " + totalColisoesHash);
    System.out.println("Colisões médias por documento: " +
        (double) totalColisoesHash / documentos.size());

    System.out.println("Fatores de carga médios das hash tables: " +
        fatoresCarga.stream().mapToDouble(Double::doubleValue).average().orElse(other: 0));

    System.out.println(x: "Árvore AVL:\n");
    avl.imprimir();

    System.out.println(x: "\n=== ANÁLISE DE COLISÕES POR DOCUMENTO ===");
    for (Documento doc : documentos) {
        TabelaHash tabela = doc.getVocabulario();
        System.out.println("Documento: " + doc.getNomeArquivo() +
                           " | Colisões: " + tabela.getColisoes() +
                           " | Tamanho da tabela: " + tabela.getTabela().length);
    }
    System.out.println(x: "====================================");
```

4. ANÁLISE EXPERIMENTAL

O controle da análise experimental é feito através de uma variável booleana no código *Main.java*. Alterando-a para *True* no código, serão impressas, via terminal, os dados para avaliação da execução.



```java
public class Main {

    Run | Debug
    public static void main(String[] args) {

        /* VARIAVEL QUE ATIVA A ANALISE EXPERIMENTAL:
            -> True para ativar
            -> False para desativar
        */
        boolean ANALISE_EXPERIMENTAL = true;
```

Para a análise experimental, vamos considerar 5 documentos de entrada. Por praticidade, os textos foram gerados por inteligência artificial (Google Gemini) e adaptados de acordo com as necessidades da análise. Abaixo, os textos utilizados:

doc1.txt
"Estrutura de dados é essencial em ciência da computação.
Aprender algoritmos é importante para resolver problemas."

doc2.txt
"Dados estruturados são essenciais para a ciência.
Aprender programação ajuda na resolução de problemas."

doc3.txt
"A computação científica envolve algoritmos e análise de dados.
Estudar estruturas de dados melhora a eficiência do código."

doc4.txt
"Estrutura de dados é essencial em ciência da computação.
Aprender algoritmos é importante para resolver problemas."

doc5.txt
"História da computação abrange desde computadores antigos até modernos sistemas.
Aprender sobre software e hardware é fundamental."

Também voltado para a praticidade, foi desenvolvido um arquivo *.bat* que contém linhas de comando que realizam os testes sequencialmente. Os comando utilizados para a análise experimental foram:

-> java Main documentos_teste 0.0 lista
-> java Main documentos_teste 0.0 topK 3
-> java Main documentos_teste 0.0 busca doc1.txt doc4.txt

Para todos os testes, foi utilizado limiar 0.0, assim pode-se observar a relação entre todos os documentos. O primeiro teste executa o modo lista, o segundo o top 3 e o último teste busca a

na avl a similaridade entre os documentos 1 e 4. Os resultados obtidos com a bateria de testes foi:

==========
TESTE 1: Executando MODO LISTA (limiar 0.0)
==========

doc1.txt <-> doc4.txt = 1,00
doc1.txt <-> doc2.txt = 0,40
doc2.txt <-> doc4.txt = 0,40
doc1.txt <-> doc3.txt = 0,34
doc3.txt <-> doc4.txt = 0,34
doc1.txt <-> doc5.txt = 0,18
doc4.txt <-> doc5.txt = 0,18
doc2.txt <-> doc3.txt = 0,17
doc2.txt <-> doc5.txt = 0,09
doc3.txt <-> doc5.txt = 0,08

Resultados salvos em resultado.txt
Rotações simples: 3
Rotações duplas: 0

=== ANÁLISE EXPERIMENTAL ===
Tempo total do programa: 66.0 ms
Tempo médio por comparação: 0.12781 ms
Tempo total em similaridade: 1.2781 ms
Tempo médio de inserção na AVL: 2.1281 ms
Total de colisões (todos os documentos): 14
Colisões médias por documento: 2.8
Fatores de carga médios das hash tables: 0.0882236227824463
Árvore AVL:

[doc1.txt <-> doc4.txt = 1,00]
[doc1.txt <-> doc2.txt = 0,40, doc2.txt <-> doc4.txt = 0,40]
[doc1.txt <-> doc3.txt = 0,34, doc3.txt <-> doc4.txt = 0,34]
[doc1.txt <-> doc5.txt = 0,18, doc4.txt <-> doc5.txt = 0,18]
[doc2.txt <-> doc3.txt = 0,17]
[doc2.txt <-> doc5.txt = 0,09]
[doc3.txt <-> doc5.txt = 0,08]

=== ANÁLISE DE COLISÕES POR DOCUMENTO ===

Documento: doc1.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc2.txt | Colisões: 2 | Tamanho da tabela: 28
Documento: doc3.txt | Colisões: 1 | Tamanho da tabela: 36
Documento: doc4.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc5.txt | Colisões: 1 | Tamanho da tabela: 34
==========
Resultados salvos em resultado.txt

==========
TESTE 2: Executando MODO TOPK (limiar 0.0, top 3)
==========

doc1.txt <-> doc4.txt = 1,00
doc1.txt <-> doc2.txt = 0,40
doc2.txt <-> doc4.txt = 0,40

Resultados salvos em resultado.txt
Rotações simples: 3
Rotações duplas: 0

=== ANÁLISE EXPERIMENTAL ===
Tempo total do programa: 72.0 ms
Tempo médio por comparação: 0.09611 ms
Tempo total em similaridade: 0.9611 ms
Tempo médio de inserção na AVL: 1.5178 ms
Total de colisões (todos os documentos): 14
Colisões médias por documento: 2.8
Fatores de carga médios das hash tables: 0.0882236227824463

Árvore AVL:

[doc1.txt <-> doc4.txt = 1,00]
[doc1.txt <-> doc2.txt = 0,40, doc2.txt <-> doc4.txt = 0,40]
[doc1.txt <-> doc3.txt = 0,34, doc3.txt <-> doc4.txt = 0,34]
[doc1.txt <-> doc5.txt = 0,18, doc4.txt <-> doc5.txt = 0,18]
[doc2.txt <-> doc3.txt = 0,17]
[doc2.txt <-> doc5.txt = 0,09]
[doc3.txt <-> doc5.txt = 0,08]

=== ANÁLISE DE COLISÕES POR DOCUMENTO ===

Documento: doc1.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc2.txt | Colisões: 2 | Tamanho da tabela: 28
Documento: doc3.txt | Colisões: 1 | Tamanho da tabela: 36
Documento: doc4.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc5.txt | Colisões: 1 | Tamanho da tabela: 34
==========
Resultados salvos em resultado.txt

==========
TESTE 3: Executando MODO BUSCA (doc1.txt vs doc4.txt)
==========

doc1.txt <-> doc4.txt = 1,00

Resultados salvos em resultado.txt
Rotações simples: 0
Rotações duplas: 0

=== ANÁLISE EXPERIMENTAL ===
Tempo total do programa: 73.0 ms
Tempo médio por comparação: 1.1792 ms
Tempo total em similaridade: 1.1792 ms
Tempo médio de inserção na AVL: 1.6489 ms
Total de colisões (todos os documentos): 14
Colisões médias por documento: 2.8
Fatores de carga médios das hash tables: 0.0882236227824463
Árvore AVL:

[doc1.txt <-> doc4.txt = 1,00]

=== ANÁLISE DE COLISÕES POR DOCUMENTO ===
Documento: doc1.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc2.txt | Colisões: 2 | Tamanho da tabela: 28
Documento: doc3.txt | Colisões: 1 | Tamanho da tabela: 36
Documento: doc4.txt | Colisões: 5 | Tamanho da tabela: 32
Documento: doc5.txt | Colisões: 1 | Tamanho da tabela: 34
==========
Resultados salvos em resultado.txt

5. RESULTADOS E DISCUSSÃO

Considerando os mesmos documentos utilizados na análise experimental e a mesma bateria de testes utilizada no tópico anterior, é importante interpretar os resultados dos testes.

No *Teste* 1 e no *Teste 2*, executado os modos *lista* e *TopK*, observa-se um comportamento completo do programa, pois todos os documentos são comparados entre si sem qualquer limiar que restrinja os resultados (limiar 0.0), a principal diferença nesses modos é a filtragem por ranking que o *TopK* faz. Os resultados foram inseridos em ordem de similaridade e a árvore realizou apenas três rotações simples para ambos testes. A raiz contém a similaridade média entre seus filhos, que do lado esquerdo possuem valores menores e do lado direito valores maiores, mostrando que a ordenação foi feita corretamente.

No *Teste 3*, como é feita a adição apenas dos documentos selecionados via terminal, a árvore AVL consiste apenas em dois documentos.

Agora, como todos os testes usaram os mesmos documentos e dois deles são idênticos, o par com maior similaridade é o *doc1* e *doc4* com 1.0 de similaridade ou 100% e o *TOP3* indicado pelo *Teste 2* é:

doc1.txt <-> doc4.txt = 1,00
doc1.txt <-> doc2.txt = 0,40
doc2.txt <-> doc4.txt = 0,40

Estes resultados são totalmente condizentes, pois como os textos dos documentos 1 e 4 são iguais, as duas posições que ocupam o *top2* e *top3* são iguais também.

Uma discussão importante a partir dos resultados é sobre as estruturas de dados, se são as mais eficientes ou se podem ser substituídas por outras estruturas sem modificar os *outputs* do projeto. A substituição da *AVL* por outra estrutura de dados, por exemplo, teria um impacto na complexidade de inserção, busca e ordenação dos resultados de similaridade. A *AVL* vai sempre garantir altura *O(log n)*, por isso mantém os custos computacionais previsíveis mesmo com grande número de comparações, o que é extremamente importante quando aumentamos a gama de documentos.

Durante as aulas, foi discutido sobre a troca por uma *Red-Black Tree*. Então, analisando, nota-se que essa mudança manteria complexidade semelhante, porém com menor custo de balanceamento e redução de rotações, assim, oferecendo desempenho prático um pouco mais constante em longas sequências de inserções, embora com maior desvio na altura real. Já o uso de uma *heap* permitiria acesso mais rápido aos maiores valores para operações como *topK*, mas comprometeria a visualização ordenada completa dos resultados, já que heaps não se preocupam com a ordenação total. Uma lista ordenada, embora simples, apresentaria custo *O(n)* por inserção, o que seria inviável para documentos com vocabulários maiores.

Assim, pode-se concluir que uma adequação da estrutura depende unicamente da prioridade de uso. Por exemplo, ordenação completa e buscas rápidas favorecem árvores balanceadas, enquanto extração repetida dos maiores valores favorece os *heaps*. Então, concluímos que a escolha das estruturas depende dos objetivos do projeto e no caso desse

projeto, as estruturas utilizadas fazem sentido com a complexidade das operações, não necessitando de uma substituição.

Outro ponto a ser discutido a partir dos resultados é a importância do pré-processamento para a execução de verificação de similaridade. Essa operação tem papel principal na redução de falsos positivos durante o cálculo de similaridade, pois sem a remoção de pontuação, normalização, padronização de maiúsculas e minúsculas e eliminação dos *tokens* irrelevantes (*stop-words)*,todas as palavras de um texto seriam contadas, alterando o vetor de frequências utilizado na comparação entre documentos. Isso causaria irregularidade, ou seja, poucas palavras seriam suficientes para modificar o valor final da similaridade, tornando o resultado inconsistente. Em suma, o pré-processamento, ao padronizar *tokens* e remover *stop-words*, garante que apenas palavras chave dos textos ocupem as tabelas *hash*, evitando falsos positivos.

6. CONCLUSÕES

Para concluir o projeto, é importante destacar a necessidade de trabalhar e entender o funcionamento de diferentes estruturas de dados em aula, pois em muitos casos elas vão ser usadas em conjunto, como foi no caso deste trabalho.

A manipulação de Tabelas *Hash* e Árvores *AVL* não fugiram muito da teoria, mas a dificuldade principal surgiu na verificação da corretude das estruturas após inserções. Como existem muitos passos de formatação e cálculos para análise de similaridade, implementar as impressões de tabelas e da estrutura da árvore necessitou muitas execuções de *debug* e muitas discussões para concluir que as inserções e rotações estavam corretas, assim como os tratamentos de colisões.

Assim, a principal dificuldade não é necessariamente a implementação do código, mas sim a visualização e compreendimento do que deve ser implementado.

Por fim, conclui-se a eficiência e funcionamento do projeto, com resultados coerentes e análises de todos os integrantes sobre os casos de teste realizados.

7. REFERÊNCIAS

ORACLE. *Java 2 Platform, Standard Edition 5.0 API Specification: java.lang.System — currentTimeMillis()*. Disponível em: https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#currentTimeMillis().

DEVMEDIA. *Leitura e escrita de arquivos de texto em Java*. Disponível em: https://www.devmedia.com.br/leitura-e-escrita-de-arquivos-de-texto-em-java/25529.

GUJ. *Árvore AVL [resolvido]*. Disponível em: https://www.guj.com.br/t/arvore-avl-resolvido/58272/3.

WIKIPEDIA. *Cosine similarity*. Disponível em: https://en.wikipedia.org/wiki/Cosine_similarity.