Trabalho Prático 3
Máquina de Busca Avançada

Luís Felipe Ramos Ferreira
2019022553

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

lframos_ferreira@outlook.com

1. Introdução

A proposta do Trabalho Prático 3, da disciplina de Estruturas de Dados, foi implementar um programa que funcione como uma máquina de busca, composta por três componentes principais, sendo eles o *crawler*, que realiza a coleta dos documentos, o *indexador*, que constrói os índices invertidos, e o *processador de consultas*, que calcula os graus de similaridade e ordena os documentos.

Mais especificamente para este trabalho, os documentos lidos estão presentes no diretório corpus, que deverá ser aberto e analisado pela máquina de busca, a fim de obter o resultado de quais documentos se assemelham mais com a consulta desejada.

2. Implementação

O programa foi desenvolvido em um ambiente operacional Windows, na linguagem C++ compilada pelo compilador G++ (versão 10.3.0) da GNU Compiler Collection.

2.1. Estrutura de dados

O programa foi construído em torno de uma lógica orientada a objetos, funcionalidade disponibilizada pela linguagem C++. Os principais mecanismos do programa, que são o Indexador e o Processador de Consultas foram implementados como classes, para que seus funcionamentos fossem mais organizados e bem implementados.

2.2. Classes

Ao todo, para a criação da máquina de busca, foram criadas 4 (quatro) classes, cada uma representando um objeto importante para o funcionamento do código.

A classe Lista foi implementada como um *template*, utilidade fornecida pela linguagem C++, e ela funciona como uma lista encadeada. Ela é utilizada para armazenar dinamicamente valores, de modo que seja fácil percorrer essa lista e acessar seus elementos. Dentre suas principais características , destacam-se os métodos *contem()*, que checa se um determinado elemento está na lista, e o método *insere()*, que insere um novo elemento ao final da lista.

Por sua vez, a classe Hash_EA foi criada para funcionar como a estrutura de hashing desejada para o armazenamento dos índices invertidos. Ela funciona utilizando a lógica de encadeamento aberto. Ou seja, é um método para evitar colisões que dispensa o

uso de apontadores, tendo como base uma tabela alocada de forma estática. É importante frisar que essa estrutura armazena elementos do tipo Termo, e o cálculo do hash é feio sobre a string que representa o termo. Em suma, o valor de hash de cada string é o resto da divisão entre o somatório dos produtos entre o valor de cada caractere na tabela ASCII e sua posição na string, pelo tamanho da tabela. Essa escolha de cálculo de hash foi feita uma vez que é um cálculo simples e eficiente para os parâmetros do trabalho.

As duas classes restantes são as que representam os principais mecanismos da máquina de busca. A classe Indexador representa o indexador de memória, e tem como principais funcionalidades: obter a lista de stopwords, isto é, as palavras frequentes que devem ser desconsideradas, obter a lista de palavras que formam o vocabulário do corpus e gerar os índices invertidos, que são os pares (ID do documento, Número de ocorrências no documento) para cada palavra do vocabulário.

A classe Processador é declarada como uma *friend class* de Indexador, para que possa acessar seus atributos privados. Ela tem como objetivo realizar o resto das operações necessárias para a máquina de busca, como ler a consulta, preencher e normalizar a lista de pesos de cada arquivo e, com isso, gerar o arquivo de saída, com os arquivos ordenados em ordem decrescente de grau de similaridade com a consulta.

2.3. Structs

A única struct criada para o programa se chama Termo, e serve para representar a estrutura de um termo que será trabalhado. Um termo é formado por uma palavra que o representa, assim como por duas listas, sendo elas uma lista que armazena os pares de (ID do documento, Número de ocorrências no documento), e a outra que armazena os pares de (ID do documento, Peso da palavra no documento), ambas para a palavra do termo.

2.4. Indexador de memória e índices invertidos

O indexador de memória tem como principal função o cálculo dos índices invertidos para todo o corpus. Para isso, em primeiro lugar, o arquivo de stopwords é percorrida e cada stopword é armazenada num arquivo auxiliar. Isso será útil para quando o indexador percorrer o corpus para armazenar, também em uma lista auxiliar, as palavras que compõem o seu vocabulário, além de contar quantos são os documentos presentes. Para o vocabulário, que é criado como um vetor, foi escolhido um tamanho de 80000, ou seja, até 80000 plavras podem ser armazenadas no vocabulário. Esse valor foi escolhido pois é importante que o maior número possível de palavras caiba no vocabulário.

Tendo estes atributos definidos, os índices invertidos são calculados ao se caminhar novamente pelo corpus, analisando cada um dos arquivos. Os índices invertidos são armazenados em uma estrutura de hashing por encadeamento aberto, como dito anteriormente.

2.5 Processador de consultas

O processador de consultas é o responsável por utilizar os índices invertidos para o cálculo dos graus de similaridade entre cada arquivo do corpus e a consulta desejada. Para isso, os passos do cálculo são feitos dentro do processador.

Em primeiro lugar, a partir da leitura do dos termos incluídos nos índices invertidos, é possível calcular o peso de cada palavra no documento, segundo a fórmula proposta, e, com isso, normalizar o vetor de pesos para cada um dos documentos presentes no corpus. Esse vetor normalizado será útil para o cálculo dos graus de

similaridade entre cada arquivo e a consulta desejada.

Os graus de similaridade são inseridos em um vetor auxiliar, após serem calculados. Cada documento possui um grau de similaridade com a consulta desejada, e, como especificado pelo trabalho, deve-se escrever no arquivo de saída os 10 documentos que mais se assemelham com a consulta. Para isso, foi necessário implementar um algoritmo de ordenação para ordenar o vetor auxiliar de graus de similaridade e, dessa maneira, ter os 10 valores de ID desejados.

O algoritmo de ordenação escolhido foi o QuickSort, por ser um método eficiente e fácil de implementar. Para garantir que, no caso em que dois graus de similaridade forem iguais, aquele com menor ID seja colocado primeiro, foi criada uma função auxiliar de comparação na implementação do QuickSort.

Por fim, o processador escreve no arquivo de saída os IDs dos 10 documentos que mais se assemelham com a consulta desejada, de acordo com os graus de similaridade calculados.

3. Análise de Complexidade

3.1. Tempo

- **Método 'insere' (classe Lista):** esse método insere um novo elemento ao final da lista. Como a cauda da lista está sempre sendo mapeada, sua complexidade de tempo é constante.

- **Método 'remove' (classe Lista):** esse método remove um elemento da lista. Sua complexidade de tempo é O(n), onde n é o número de elementos que devem ser percorridos na lista até encontrar o que deve ser removido.

- **Método 'contem' (classe Lista):** esse método percorre a lista para checar se determinado elemento está contido nela, logo, sua complexidade é linear.

- **Método 'limpa' (classe Lista):** esse método percorre a lista deletando cada um dos seus elementos, portanto, sua complexidade de tempo é linear.

- **Método 'tamanho' (classe Lista):** esse método apenas retorna o valor do atributo de tamanho da lista e, por isso, sua complexidade de tempo é constante.

- **Método 'vazia' (classe Lista):** esse método apenas retorna se a lista está vazia ou não, baseado no tamanho dela, logo, sua complexidade é constante.

- **Método 'conta' (classe Lista):** esse método percorre a lista e conta quantas vezes determinado elemento aparece e, por isso, sua complexidade de tempo é linear.

- **Método 'Hash' (classe Hash_EA):** esse método percorre uma string para calcular seu valor de hash, portanto, sua complexidade de tempo é O(n), onde n é o tamanho da string.

- **Método ‘insere’ (classe Hash_EA):** esse método, no melhor caso, insere o elemento na tabela na primeira posição checada, em tempo constante. No pior caso, toda a tabela deve ser percorrida até que uma posição disponível seja encontrada, gerando uma complexidade de tempo linear.

- **Método ‘remove’ (classe Hash_EA):** esse método procura o elemento a ser removido da tabela. No melhor caso, esse elemento é o primeiro procurado, resultando em tempo constante. No pior caso, a tabela inteira é percorrida, o que gera uma complexidade de tempo linear.

- **Método ‘contem’ (classe Hash_EA):** esse método checa se o elemento está na tabela. No melhor caso, esse elemento é o primeiro procurado, resultando em tempo constante. No pior caso, a tabela inteira é percorrida, o que gera uma complexidade de tempo linear.

- **Método ‘pesquisa’ (classe Hash_EA):** esse método procura o elemento desejado na tabela. No melhor caso, esse elemento é o primeiro procurado, resultando em tempo constante. No pior caso, a tabela inteira é percorrida, o que gera uma complexidade de tempo linear.

- **Método ‘at’ (classe Hash_EA):** esse método retorna o termo presente numa determinada posição na tabela de hashing, portanto, sua complexidade de tempo é constante.

- **Método ‘getTamanho’ (classe Hash_EA):** esse método retorna o tamanho da tabela de hashing, logo sua complexidade de tempo é constante.

- **Método ‘getPalavra’ (struct Termo):** esse método retorna o valor do atributo palavra do termo, portanto sua complexidade de tempo é constante.

- **Método ‘getNumDeOcorrencias’ (struct Termo):** esse método percorre o vetor de pares de (ID do documento, Número de ocorrências no documento) até encontrar o ID desejado, para retornar o número de ocorrências. Logo, sua complexidade de tempo é linear.

- **Método ‘getPeso’ (struct Termo):** esse método percorre o vetor de pares de (ID do documento, Peso da palavra no documento) até encontrar o ID desejado, para retornar o peso da palavra no ID. Logo, sua complexidade de tempo é linear.

- **Método ‘getTamanhoParIDnumDeOcorrencias’ (struct Termo):** esse método apenas retorna o valor do atributo que controla quantos elementos existem no vetor de pares de (ID do documento, Número de ocorrências no documento), logo, sua complexidade de tempo é constante.

- **Método ‘insereParIDnumDeOcorrencias’ (struct Termo):** esse método apenas insere no vetor de pares de (ID do documento, Número de ocorrências no documento) , na posição correta, o elemento desejado. Portanto, sua complexidade de tempo é constante.

- **Método ‘contemParIDnumDeOcorrencias’ (struct Termo):** esse método percorre a lista de pares de (ID do documento, Número de ocorrências no documento) para checar se ele contém o par desejado. Logo, sua complexidade de tempo é linear.

- **Método ‘insereParIDPeso’ (struct Termo):** esse método apenas insere no vetor de pares de (ID do documento, Peso da palavra no documento) , na posição correta, o elemento desejado. Portanto, sua complexidade de tempo é constante.

- **Método ‘contemID’ (struct Termo):** esse método percorre a lista de pares de (ID do documento, Número de ocorrências no documento) para checar se ele contém o ID desejado. Logo, sua complexidade de tempo é linear..

- **Função ‘swap’ (QuickSort):** essa função utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Função ‘quickSort’ (QuickSort):** essa função utiliza a função ordena para todo o vetor que deve ser ordenado. Ela possui, em caso médio, uma complexidade de tempo O(n log(n)).

- **Método ‘getStopWords’ (classe Indexador):** esse método percorre o arquivo que contém as stopwords, para armazená-las em uma lista. Logo, sua complexidade de tempo é O(n), onde n é a quantidade de stopwords no arquivo.

- **Método ‘isStopWord’ (classe Indexador):** esse método chama o método ‘contem’ da classe Lista, logo, sua complexidade de tempo é linear.

- **Método ‘getVocabulario’ (classe Indexador):** esse método percorre por cada um dos n documentos do diretório corpus e, em cada documento, percorre por suas m (em média) palavras. Para cada uma dessas palavras, é necessário checar se ela é uma stopword ou se já está presente no vocabulário, analisando suas respectivas listas, com tamanhos s e v. Portanto, sua complexidade de tempo é O(nm(s + v)).

- **Método ‘getIndicesInvertidos’ (classe Indexador):** esse método percorre por cada um dos n documentos do diretório corpus e, em cada documento, percorre por suas m (em média) palavras. Para cada uma dessas palavras, é necessário checar se ela é uma stopword, em sua lista de tamanho s. Nesse caso, a complexidade de tempo é O(nms). Caso contrário, é necessário percorrer o conteúdo do arquivo e contar quantas vezes a palavra aparece. Após isso, é checado se a palavra já está presente nos índices invertidos, de tamanho i. Se não, a complexidade de tempo será O(nm(m + i)). Se sim, um novo par é inserido nos índices invertidos, na posição correta. Da mesma maneira, nesse caso, a complexidade de tempo será O(nm(m + i)).

- **Função ‘removeCaractereSeparadoresTexto’:** esse método utiliza a função replace(), da classe string, e por isso sua complexidade de tempo é linear.

- **Função ‘textoCaixaBaixa’:** esse método utiliza a função transform, da biblioteca algorithm, e por isso sua complexidade de tempo é linear.

- **Função ‘getIDDocumento’:** essa função percorre um laço de tamanho igual ao número n de caracteres na string que representa o ID do documento, portanto, sua complexidade de tempo é O(n).

- **Função ‘contaPalavraTexto’:** essa função percorre cada uma das n palavras de um texto e, portanto, possui complexidade de tempo O(n).

- **Método ‘calculaPesoPalavra’ (classe Processador):** esse método chama o método getNumDeOcorrencias() da struct Termo, logo, sua complexidade de tempo é linear.

- **Método ‘normalizaVetorDePesos’ (classe Processador):** esse método percorre o vetor de IDs de documentos, de tamanho n, e, dentro desse laço, percorre um vetor de tamanho m do hash de índice invertido. Dentro desse segundo laço, o método calculaPesoPalavra() é chamado, que possui complexidade de tempo linear para o tamanho p de pares (ID do documento, Número de ocorrências do documento). Dessa forma, a complexidade de tempo dessa função é cúbica, isto é, O(nmp).

- **Método ‘preencheVetorDeGrausDeSimilaridade’ (classe Processador):** esse método percorre o vetor de IDs de documentos, de tamanho n, e, dentro desse laço, percorre cada uma das m palavras na consulta. Para cada palavra, é chamado o método pesquisa() da classe Hash_EA, que possui complexidade de tempo linear para o tamanho p da tabela de hashing. Dentro do segundo laço, é chamada a função getPesoNormalizado(), de complexidade de tempo linear para o tamanho n. Portanto, sua complexidade de tempo é cúbica, isto é, $O(n(mp + n))$ ou $O(nmp + n^2)$

- **Método ‘getPesoNormalizado’ (classe Processador):** esse método percorre o vetor de pesos normalizados, de tamanho n, em busca do ID desejado. Portanto, sua complexidade de tempo é linear.

- **Método ‘geraArquivoDeSaida’ (classe Processador):** esse método utiliza o algoritmo de ordenação QuickSort e, por isso,possui complexidade de tempo O(nlog(n)).

3.2. Espaço

- **Método ‘insere’ (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘remove’ (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘contem’ (classe Lista):** esse método realiza todas as suas

operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'limpa' (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'tamanho' (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'vazia' (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'conta' (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'Hash' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'insere' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'remove' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'contem' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'pesquisa' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'at' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getTamanho' (classe Hash_EA):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getPalavra' (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getNumDeOcorrencias’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getPeso’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getTamanhoParIDnumDeOcorrencias’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘insereParIDnumDeOcorrencias’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘contemParIDnumDeOcorrencias’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘insereParIDPeso’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘contemID’ (struct Termo):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘swap’ (QuickSort):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘quickSort’ (QuickSort):** esse método realiza suas operações sobre a lógica de dividir para conquistar. No pior caso, utiliza um espaço adicional de ordem n, então, possui complexidade de espaço O(n).

- **Método ‘getStopWords’ (classe Indexador):** esse método cria uma lista de tamanho n, contendo as stopwords lidas do arquivo. Portanto, sua complexidade de espaço é O(n).

- **Método ‘isStopWord’ (classe Indexador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getVocabulario’ (classe Indexador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getIndicesInvertidos’ (classe Indexador):** esse método realiza

todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função 'removeCaractereSeparadoresTexto':** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função 'textoCaixaBaixa':** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função 'getIDDocumento':** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função 'contaPalavraTexto':** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'calculaPesoPalavra' (classe Processador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'normalizaVetorDePesos' (classe Processador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'preencheVetorDeGrausDeSimilaridade' (classe Processador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getPesoNormalizado' (classe Processador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'geraArquivoDeSaida' (classe Processador):** esse método utiliza o algoritmo de ordenação QuickSort e, por isso, sua complexidade de espaço é O(n).

4. Testes

Os principais testes foram realizados sobre as 10 consultas disponibilizadas pelos professores para o diretório colecao_small, que possui 202 documentos. Os resultados, para cada uma das consultas, são os seguintes:

- Consulta 1

  14681 13864 8877 14259 14850 8763 7425 15323 4536 1599

- Consulta 2

14506 8611 239 490 6125 10856 15636 14489 11714 11220

- Consulta 3

  3180 3647 13678 5384 5332 6187 9917 3894 17571 12359

- Consulta 4

  13682 12804 15636 14843 12509 14128 11002 3629 5174 10002

- Consulta 5

  5668 16965 16213 17684 16870 8930 15264 5548 1291 8140

- Consulta 6

  17801 649 16722 17593 16672 18547 16705 16661 8082 2211

- Consulta 7

  7442

- Consulta 8

  16281

- Consulta 9

  16528 1253 2353 8764 16360 16361 18111 1499 8505 11335

- Consulta 10

  2353 70 321 64 315 1246 4871 8764 4109 5237

5. Considerações importantes

É relevante citar algumas coisas acerca do programa criado. Ao tentar realizar a análise experimental, o meu computador sequer terminou a execução do programa para o diretório colecao_full disponibilizado pelos professores, provavelmente devido às suas especificações de processamento, dentre outras questões de hardware, uma vez que é um computador velho.

Devido a isso, a análise experimental não foi concluída da maneira desejada e não foi incluída na documentação. No entanto, considerei importante citar essa situação, uma vez que não pude executar e analisar meu código para diretórios com grandes quantidades de arquivos.

6. Estratégias de robustez

Para tratar falhas e deixar o código mais robusto, foi utilizada a biblioteca ‘msgassert.h’, disponibilizada pelos professores. Essa biblioteca define duas ‘macros’ para tratar os possíveis erros encontrados no uso das funções do programa.

Ela foi utilizada em funções e métodos onde foi julgado necessário garantir que parâmetros de entrada fossem dados da forma esperada. Ela também foi utilizada para garantir que os arquivos utilizados na implementação fossem abertos corretamente.

7. Conclusão

O trabalho prático teve como propósito criar uma máquina de busca, a qual coleta os documentos desejados, por meio de um *crawler*, os organiza de acordo com os índices invertidos, por meio do indexador de memória, e ordena os documentos de acordo com aqueles que mais se assemelham à consulta realizada, por meio do processador de consultas..

Para a implementação deste, cada uma das partes da máquina de busca foi identificada como um processo de implementação. Em uma classe, o indexador foi definido e, junto a ele, o *crawler* realiza seus comandos de coleta dos documentos. O indexador, por sua vez, calcula os índices invertidos que serão utilizados posteriormente, os quais são armazenados em uma estrutura de hashing por encadeamento aberto.

O processador utiliza, em uma classe criada para si, o valor destes índices invertidos, para então calcular o grau de similaridade entre cada um dos documentos do diretório corpus e a consulta desejada. Essa operação leva em consideração o peso de cada palavra do vocabulário em seus respectivos documentos, assim como a normalização do vetor de pesos de cada documento.

O trabalho trata de temas muito importantes para a computação, principalmente no que concerne ao funcionamento de uma máquina de busca. A internet, de maneira geral, funciona em volta deste mecanismo, e entender como ele funciona, mesmo que de forma básica e inicial, é importante para compreender o paradigma geral do tema. Tendo isso em mente, salienta-se que as estratégias tomadas para a execução do trabalho foram tomadas visando a maior eficiência possível para a realidade exposta.

Dessa maneira, conclui-se que a implementação deste trabalho foi extremamente importante para a fixação e análise de diversos conceitos importantes, assim como supracitado sobre o mecanismo da máquina de busca. Mais especificamente sobre pequenas partes da implementação, alguns conceitos foram revisados e retrabalhados, como algoritmos de ordenação, leitura e escrita em arquivos e manipulação de vetores alocados dinamicamente.

8. Referências bibliográficas

MIZRAHI, Victorine Viviane. Treinamento em linguagem C. 2ª Edição. Editora Pearson.

DAEMONIO. Usando As Funções getopt() e getopt_long() Em C. daemoniolabs. Disponível em: <https://daemoniolabs.wordpress.com/2011/10/07/usando-com-as-funcoes-getopt-e-getopt_long-em-c/>. Acesso em: 15/02/2022.

DOCS-MICROSOFT.Value categories, and references to them.microsoft. Disponível em: <https://docs.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/cpp-value-categories>.Acesso em: 15/02/2022.

CP-ALGORITHMS.String Hashing.cp-algorithms. Disponível em: <https://cp-algorithms.com/string/string-hashing.html#:~:text=Calculation%20of%20the%20hash%20of%20a%20string,-The%20good%20and&text=It%20is%20called%20a%20polynomial,alphabet%2C%20is%20a%20good%20choice>. Acesso em: 16/02/2022.

UNESP. Tutorial: introdução ao uso do aplicativo Gnuplot.UNesp. Disponível em: <http://www2.fct.unesp.br/docentes/carto/galo/web/gnuplot/pdf/2017_Galo_Gnuplot_Tutorial.pdf>. Acesso em: 18/02/2022.

GNUPLOT.Gnuplot homepage. gnuplot. Disponível em: <http://www.gnuplot.info/>. Acesso em: 19/02/2022.

DROZDEK, Adam. Estrutura de dados e algoritmos em c++. 4ªEdição. Editora Cengage Learning.

LEFTICUS.Style: C++ best practices.lefticus.Disponível em: <https://lefticus.gitbooks.io/cpp-best-practices/content/03-Style.html>.Acesso em: 19/02/2022.

9. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize comando *make* para compilar os arquivos e gerar o 'target' do programa.
- Para utilizar o programa, digite no terminal ‘bin/programa’, seguido das opções de linha de comando. As opções são as seguintes:
  -i (Nome do arquivo de entrada)
  -o (Nome do arquivo de saída)
  -c (Nome do diretório corpus)
  -s (Nome do arquivo de stopwords)
- Exemplo de execução: bin/programa -i entrada.txt -o saida.txt -c corpus -s stopwords.txt
- Para apagar os arquivos ‘object’ e o ‘target’, utilize no terminal o comando make clean.