Trabalho Prático: Máquina de Busca Avançada

Gabriel Teixeira Carvalho

1. Introdução

Esta documentação lida com o problema de implementar uma máquina de busca. O objetivo principal desta tarefa é criar um programa que, dada uma série de documentos (corpus) e palavras a serem consultadas nos documentos, consiga tratar os textos presentes nos documentos e ordenar os documentos de acordo com a similaridade deles com os termos consultados, retirando palavras que não contenham muita informação (stopwords) dos documentos e consultas no processo. Além disso, outro foco importante foi a revisão de conceitos de hash, ordenação, estruturas de dados, classes, desempenho e robustez de código, sendo muito aplicados ao longo da implementação. Para resolver o problema citado, foi utilizado um índice invertido (hash) que permite encontrar em qual documento está uma palavra especificada de maneira eficiente.

A seção 2 desta documentação trata mais a fundo sobre como o programa foi implementado, enquanto na seção 3 é apresentada uma análise da complexidade temporal e espacial do trabalho. Já na seção 4, são apresentados aspectos de robustez e proteção contra erros de execução. A seção 5 destrincha o comportamento do programa em termos de tempo e espaço através de diversos experimentos realizados após a implementação do programa. Por fim, a seção 6 conclui e sumariza tudo que será falado nesta documentação, sendo seguida por referências bibliográficas utilizadas na confecção do código e por um manual de compilação e execução do programa.

2. Método

2.1. Ambiente de desenvolvimento

O programa foi implementado na linguagem C++, compilado pelo G++, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL2), com um processador Intel Core i7-6500U (2.50GHz, 4 CPUs) e 8192 MB RAM.

2.2. Organização, Arquivos e Funcionamento

O código está organizado em 4 diretórios na pasta raiz:

- `obj`: Esta pasta contém os arquivos `.cpp` da pasta `src` traduzidos para linguagem de máquina pelo compilador GCC e com a extensão `.o` (object).
- `bin`: Contém o arquivo executável `main`, que consiste nos arquivos .o da pasta `obj` compilados em um só arquivo pelo GCC.
- `include`: Contém os arquivos de declaração (contrato) das classes utilizados no programa, com seus atributos e métodos especificados dentro

de arquivos da extensão `.hpp`, que serão incluídos e implementados nos arquivos de extensão `.cpp`:

- `resultado.hpp`: Define o TAD Resultado que guarda a similaridade de um documento e o ID dele.
- `termoVocabulario.hpp`: Define o TAD termoVocabulario que guarda um termo e a frequência dele em um documento.
- `ocorrencia.hpp`: Define o TAD Ocorrencia que guarda um documento e sua frequência para utilizar no índice invertido.
- `termoIndice.hpp`: Define o TAD termoIndice que guarda um termo e uma ListaEncadeadaOcorrencia.
- `processadorDeDocumentos.hpp`: Define a classe ProcessadorDeDocumentos, com métodos para tratar os textos de um documento (deixando apenas caracteres alfabéticos minúsculos), processar o corpus inteiro e contar o número de termos em um documento.
- `vocabulario.hpp`: Define a classe Vocabulario, com métodos para criar o vocabulário de termos e frequências de um documento e verificar se uma palavra é uma stopword.
- `quicksort.hpp`: Define a classe QuickSort, com métodos para ordenação de um vetor de resultados e um atributo que guarda o tamanho do vetor a ser ordenado.
- `listaEncadeadaOcorrencia.h`: Define a classe `ListaEncadeadaOcorrencia` que possui células de Ocorrencia encadeadas através de ponteiros. Cada célula representa uma ocorrência (id e frequência) de uma palavra em um documento e aponta para a próxima célula na lista, formando uma lista de Ocorrencias. Essa classe possui um método que permite inserir uma Ocorrencia no final.
- `listaEncadeadaTermoIndice.h`: Define a classe `ListaEncadeadaTermoIndice` que possui células de TermoIndice encadeadas através de ponteiros. Cada célula representa um termo do índice invertido, com sua palavra e suas ocorrências, e aponta para a próxima célula na lista, formando uma lista de TermoIndices. Essa classe possui um método que permite inserir um TermoIndice no final.
- `processadorDeConsultas.hpp`: Define a classe `ProcessadorDeConsultas` que possui métodos que permitem calcular a norma dos documentos presentes no corpus, processar uma consulta, ordenar os resultados da consulta e imprimí-los em um arquivo de saída.
- `indiceInvertido.hpp`: Define a classe IndiceInvertido, formada por uma tabela de Hash que permite inserir inserir e recuperar as ocorrências de um termo de maneira eficiente, utilizando de uma função hash

    - `opcoesMain.hpp`: Define a classe OpcoesMain com atributos e um método que permitem receber parâmetros pela linha de comando.
    - `memlog.h`: Define um TAD que gerencia o registro dos acessos à memória e o registro do desempenho, através de métodos utilizados sempre que há leitura ou escrita em memória.
    - `msgassert.h`: Define macros que conferem condições necessárias para o funcionamento correto do programa, abortando a execução ou imprimindo um aviso caso estas condições sejam descumpridas.

- `src`: Contém as implementações das classes declarados nos arquivos `.hpp` da pasta `include` em arquivos `.cpp`; Nestes arquivos estão programadas as regras, a lógica e a robustez contra erros que é o alicerce de todo o programa:
    - `processadorDeDocumentos.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `processadorDeDocumentos.hpp`. Métodos mais importantes:
        - `processaDocumento`: processa um documento, abrindo o arquivo original, pegando cada palavra, tirando tudo que não é letra dela, inserindo em um arquivo novo, deletando o arquivo original quando todas as palavras são transferidas e renomeando o arquivo novo.
        - `processaCorpus`: percorre todos os arquivos do corpus processando cada documento.

    - `vocabulario.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `vocabulario.hpp`. Métodos mais importantes:
        - `adicionaTermoVocabulario`: procura o termo no vocabulário e aumenta sua frequência se existir, criando o termo no vocabulário se não existir.
        - `criaVocabularioDocumento`: adiciona cada termo de um documento no vocabulário.
        - `eStopword`: retorna se o termo é uma stopword ou não.
    - `quicksort.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `quicksort.hpp`. Métodos mais importantes:
        - `particao`: Particiona um vetor de resultados a partir de um pivô escolhido. Elementos maiores do que o pivô são colocados à sua esquerda e elementos menores vão para sua direita. O pivô escolhido é a mediana de 3 elementos do vetor para evitar que o QuickSort caia no seu pior caso.
        - `ordena`: Chama o método `particao` e o próprio `ordena` para ordenar o vetor de resultados recursivamente.

    - `listaEncadeadaOcorrencia.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo

`listaEncadeadaOcorrencia.hpp`. Métodos mais importantes:
    - `insereFinal`: Insere uma Ocorrencia no final da lista.

- `listaEncadeadaTermoIndice.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `listaEncadeadaTermoIndice.hpp`. Métodos mais importantes:
    - `insereFinal`: Insere um TermoIndice no final da lista.

- `processadorDeConsultas.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `processadorDeConsultas.hpp`. Métodos mais importantes:
    - `calculaNormaDocumentos`: percorre o índice invertido, calculando o tf x idf (Recuperação de Informação) de cada ocorrência para normalizar o vetor de resultados e depois tirando a raiz quadrada de cada valor achado.
    - `processaConsulta`: percorre o arquivo de consulta passado como parâmetro pela linha de comando, le cada termo e soma o tf x idf para cada documento que o termo aparece, normaliza os resultados dividindo cada valor pela norma do documento correspondente.
    - `ordenaResultados`: ordena o vetor de resultados através da classe Quicksort, deixando o vetor em ordem decrescente das similaridades e desempatando pelo ID do documento em ordem crescente.
    - `imprimeResultados`: imprime os 10 documentos mais similares, ou menos se não houverem 10.

- `indiceInvertido.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `indiceInvertido.hpp`. Métodos mais importantes:
    - `calculaHash`: calcula a função hash do termo passado como parâmetro, de forma a evitar colisões entre 2 termos diferentes passadas como parâmetro.
    - `insere`: chama a função calculaHash para o termo passado e insere o termo na ListaEncadeadaTermoIndice na posição retornada pela função calculaHash e a Ocorrencia em sua ListaEncadeadaOcorrencia.
    - `pesquisa`: percorre a ListaEncadeadaTermoIndice na posição de hash calculada até achar o termo passado como parâmetro, por conta das possíveis colisões. Então, retorna a ListaEncadeadaOcorrencia encontrada.
    - `criaIndice`: percorre todos os documentos do corpus, criando seu vocabulário e, para cada termo do vocabulário, criando uma Ocorrencia e inserindo na tabela de hash.

- `opcoesMain.cpp`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `opcoesMain.hpp`. Métodos mais importantes:
  - `parse_args`: recebe os argumentos necessários para a execução do programa da linha de comando e atribui os valores nas variáveis.

- `main.cpp`: Este arquivo implementa a função principal do programa, responsável por controlar o fluxo de execução do código. Esse arquivo é responsável pela instanciação dos objetos das classes implementadas e chamada dos métodos das classes para que a busca seja feita.
- `memlog.cpp`: Neste arquivo está presente a implementação do TAD que registra os acessos de memória e desempenho de tempo. Métodos mais importantes:
  - `leMemLog`: Registra em um arquivo uma operação de leitura de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `escritaMemLog`: Registra em um arquivo uma operação de escrita de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `defineFaseMemLog`: Separa as fases de execução do programa, fundamental para a análise experimental.

Além disso, há um arquivo `makefile`, responsável pela compilação modularizada do programa, permitindo compilar somente os módulos necessários, ao invés de recompilar todos os arquivos a cada mudança.

2.3. Detalhes de implementação

A estrutura de dados fundamental para o trabalho, utilizadas para pesquisar nos documentos, foi uma tabela de hash. Os vetores que foram alocados dinamicamente foram devidamente desalocados para evitar vazamentos de memória. Outra estrutura importante é o TAD `memlog_tipo`, que guarda informações necessárias para o registro de performance.

As operações que o usuário pode executar no programa e suas opções de linha de comando são as seguintes:

`'-i <arquivo>'`	Define o caminho do arquivo que contém as consultas a serem feitas
`'-o <arquivo>'`	Define o caminho do arquivo de saída
`'-c <pasta>'`	Define o caminho da pasta de documentos (corpus)
`'-s <arquivo>'`	Define o caminho do arquivo de stopwords
`'-p <arquivo>'`	Define o caminho do arquivo de registro de desempenho

`'-l'`	Define se deve ser registrados todos os acessos a memória feitos pelas funções `leMemLog` e `escreveMemLog` no arquivo de registro de desempenho.

Por fim, cabe ressaltar que todas as passagens de matrizes por parâmetro para as funções e métodos é feita por referência, passando o endereço ao invés de copiar a matriz em questão. Isto economiza grandes quantidades de memória a cada chamada de função.

3. Análise de Complexidade

Nesta seção será analisada a complexidade tanto de tempo como de espaço das funções e métodos descritos acima. Sendo N = número de termos em cada documento, M = tamanho de cada termo, P = número de documentos e assumindo que as operações de leitura e escrita em arquivo sejam O(1) em complexidade de tempo:

Método `ProcessadorDeDocumentos::processaDocumento` - complexidade de tempo:

Esse método percorre o documento lendo um termo por vez, percorre o termo um caractere por vez e depois escreve o termo em um arquivo novo. Portanto, esse método é O(N*M). Como as strings são pequenas, na verdade é O(N).

Método `ProcessadorDeDocumentos::processaDocumento` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `ProcessadorDeDocumentos::processaCorpus` - complexidade de tempo:

Esse método chama o método processaDocumento para cada documento do corpus, logo, é O(N*P).

Método `ProcessadorDeDocumentos::processaCorpus` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `Vocabulario::adicionaTermoVocabulario` - complexidade de tempo:

Esse método percorre o vocabulário inteiro e insere o termo no final no pior caso, que acontece quando o termo não é encontrado. Logo, esse método é O(K), sendo K o tamanho do vocabulário.

Método `Vocabulario::adicionaTermoVocabulario` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `Vocabulario::criaVocabularioDocumento` - complexidade de tempo:

Esse método lê cada palavra válida de um documento e a insere no vocabulário. Se cada palavra for inédita, o método adicionaTermoVocabulário cairá sempre no pior caso e esse método será $O(1 + 2 + 3 + 4 + ... + N) = O(N^2)$. Se a mesma palavra for repetida até o final, esse método será $\Theta(N)$.

Método `Vocabulario::criaVocabularioDocumento` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `Vocabulario::eStopword` - complexidade de tempo:

Esse método percorre a lista de stopwords comparando o termo passado por parâmetro com cada stopword. Logo, é Θ (K), sendo K o tamanho da lista de stopwords.

Método `Vocabulario::eStopword` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `QuickSort::particao` – complexidade de tempo:

Esse método percorre o vetor de resultados comparando e trocando elementos que estão antes e depois do pivô, organizando o vetor, até que o índice que vem da esquerda encontre o índice que vem da direita. Portanto, esse método percorre todas as posições do vetor de tamanho N, sendo assim, $\Theta(N)$.

Método `QuickSort::particao` – complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de resultados é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `QuickSort::ordena` - complexidade de tempo:

Esse método chama o método `particao` e também chama o próprio `ordena` recursivamente uma vez em cada partição do vetor de resultados. No melhor caso, no qual as duas partes têm sempre tamanhos iguais, esse método tem equação de recorrência $T(N) = 2(N/2) + N$, que possui solução $\Theta(N\log N)$. No caso médio, é demonstrado que o QuickSort também possui complexidade $\Theta(N\log N)$. No pior caso, quando o pivô escolhido é repetidamente uma das extremidades do vetor, o QuickSort é $\Theta(N^2)$, porém, esse caso é evitado no programa ao definir o pivô como a mediana de uma amostra de três valores do vetor. Portanto, esse método possui complexidade $\Theta(N\log N)$.

Método `QuickSort::ordena` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada e o vetor de resultados é passado como parâmetro por referência, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `ListaEncadeadaOcorrencia::insereFinal` - complexidade de tempo:

Esse método insere uma célula nova no final, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(1).

Método `ListaEncadeadaOcorrencia::insereFinal` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ListaEncadeadaTermoIndice::insereFinal` - complexidade de tempo:

Esse método insere uma célula nova no final, ajusta os atributos e ponteiros das células e o tamanho da lista. Sua complexidade é constante, Θ(1).

Método `ListaEncadeadaTermoIndice::insereFinal` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de Lista, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ProcessadorDeConsultas::calculaNormaDocumentos` - complexidade de tempo:

Esse método percorre cada posição da tabela de hash (N*P) e para cada TermoIndice (tamanho K = número de colisões), percorre sua ListaEncadeadaOcorrencia (tamanho L = número de ocorrências), calculando o tf x idf, somando em um vetor de tamanho P e depois tirando a raiz quadrada de cada valor do vetor. Logo, a complexidade desse método é O(N*P*K*L + P) mas se aproxima de O(N*P*L) pois o número de colisões é muito baixo.

Método `ProcessadorDeConsultas::calculaNormaDocumentos` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de ListaEncadeadaTermoIndice e um ponteiro para Célula de

ListaEncadeadaoOcorrencia mas as desaloca liberando o espaço, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, Θ(1).

Método `ProcessadorDeConsultas::processaConsulta` - complexidade de tempo:

Esse método percorre a consulta lendo termos e, para cada termo (J), chama o método Vocabulario::eStopword (O(K), K = tamanho da lista de stopwords) e o método indiceInvertido::pesquisa, além disso, para cada Ocorrencia retornada pela pesquisa (L), calcula seu tf x idf. Além disso, divide cada elemento do vetor de resultados (P) pelo valor correspondente do vetor de norma dos documentos. Logo, sua complexidade é O(J*(K + L) + P).

Método `ProcessadorDeConsultas::processaConsulta` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de ListaEncadeadaOcorrencia que é desalocado e um ponteiro de ListaEncadeadaOcorrencia que é preenchido com os resultados da pesquisa e também é limpado. Portanto, a complexidade assintótica de espaço é Θ(1).

Método `ProcessadorDeConsultas::ordenaResultados` - complexidade de tempo:

Esse método chama o método `QuickSort::ordena` que é Θ(PlogP).

Método `ProcessadorDeConsultas::ordenaResultados` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `ProcessadorDeConsultas::imprimeResultados` - complexidade de tempo:

Esse método imprime no arquivo de saída cada elemento do vetor de resultados, que corresponde a cada documento, logo é Θ(P).

Método `ProcessadorDeConsultas::imprimeResultados` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é Θ(1).

Método `IndiceInvertido::calculaHash` - complexidade de tempo:

Esse método percorre o termo calculando um valor para cada caractere, logo é Θ(M). Como as strings são pequenas, é Θ(1).

Método `IndiceInvertido::calculaHash` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Método `IndiceInvertido::insere` - complexidade de tempo:

Esse método chama o calculaHash, percorre a ListaEncadeadaTermoIndice (L = número de colisões) na posição da tabela calculada pelo calculaHash até achar o termo correspondente a ser inserido ou, caso não seja encontrado, inserindo no final da lista. Logo, é O(M + L). Porém, como os termos e o número de colisões são baixos, ele pode ser considerado constante, $\Theta(1)$.

Método `IndiceInvertido::insere` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de ListaEncadeadaTermoIndice, independentemente do tamanho da entrada. Portanto, a complexidade assintótica de espaço é constante, $\Theta(1)$.

Método `IndiceInvertido::pesquisa` - complexidade de tempo:

Esse método chama o calculaHash, percorre a ListaEncadeadaTermoIndice (L = número de colisões) na posição da tabela calculada pelo calculaHash até achar o termo correspondente a ser retornado ou, caso não seja encontrado, retornando uma ListaEncadeadaOcorrencia vazia. Logo, é O(L). Porém, como o número de colisões é baixo, ele pode ser considerado constante, $\Theta(1)$.

Método `IndiceInvertido::pesquisa` - complexidade de espaço:

Esse método aloca espaço para um ponteiro para Célula de ListaEncadeadaTermoIndice que é desalocado e um ponteiro de ListaEncadeadaOcorrencia vazio. Portanto, a complexidade assintótica de espaço é $\Theta(1)$.

Método `IndiceInvertido::criaIndice` - complexidade de tempo:

Esse método percorre o corpus e, para cada arquivo, cria seu vocabulário, para cada termo do vocabulário, insere-o no hash e desaloca o vocabulário. Logo, é $O(P*N^2)$.

Método `IndiceInvertido::criaIndice` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, portanto, sua complexidade assintótica de espaço é $\Theta(1)$.

Função `main` - complexidade de tempo:

Essa função controla a execução do programa. Ela chama os métodos processaCorpus (O (N*P)), criaIndice (O ($P*N^2$)), calculaNormaDocumentos (O (N*P*L)), processaConsulta (O(J*(K + L) + P)), ordenaResultados (O(PlogP)) e imprimeResultados (O(P)). Portanto, o programa tem complexidade O($N*P + P*N^2$

+ N*P*L + J*(K + L) + P + PlogP + P). Como o número de documentos domina sobre todos os fatores, esse programa se aproxima de O(PlogP).

Função `main` - complexidade de espaço:

Essa função cria um vetor do tamanho do número de documentos, logo é Θ(P).

Método `memlog::leMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::leMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::escreveMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::escreveMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Método `memlog::defineFaseMemLog` - complexidade de tempo:

Esse método realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Método `memlog::defineFaseMemLog` - complexidade de espaço:

Esse método requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Além disso, para o registro dos acessos à memória, a cada operação de leitura ou escrita será realizada uma operação de impressão de complexidade O(1). Essa opção está desativada por padrão no código, mas pode ser ativada facilmente para que as análises sejam feitas. Isso não muda a ordem de complexidade da operação escolhida, porém, aumenta a constante que multiplica a função de complexidade da operação. Portanto, essa opção pode reduzir bastante a performance do código.

4. Estratégias de Robustez

Ao longo do código, a macro `erroAssert é utilizada para tratar erros`. A macro assegura certas condições importantes para o programa, gerando uma mensagem e abortando a execução do programa caso a condição não seja atendida. Os mecanismos de programação defensiva e robustez utilizados foram:

- Asserção de nome do arquivo passado por parâmetro – aborta o programa se o nome do arquivo passado por parâmetro não for especificado.

- Asserção de abertura de arquivo – aborta o programa se o arquivo de comandos ou de saída não forem abertos.
- Asserção de parâmetro numérico válido – aborta o programa se algum parâmetro numérico tiver um valor inválido.
- Asserção de consulta ou pesquisa sem resultados – emite um aviso se nenhum documento tiver a palavra consultada ou pesquisada.

5. Análise Experimental

Essa seção compila uma série de experimentos de diferentes tipos que analisam a performance do programa por meio dos registros do TAD `memlog_tipo`.

5.1. Análise de tempo

Esse experimento tem como objetivo medir o tempo de execução ao performar as operações implementadas no código com diferentes cargas de trabalho

O tempo é obtido ao subtrair o tempo inicial do tempo final, os quais estão presentes no arquivo de registro de desempenho. Foram geradas 12 cargas de trabalho que variam na quantidade de documentos, de 5 até 10240. Foi feita a mesma bateria de testes para cada carga: o programa é executado 10 vezes e a média de tempo entre as execuções é calculada. Ao analisar os dados obtidos e traçando um gráfico com a ajuda do programa GNUPLOT, percebe-se que o programa tem um comportamento que condiz com a análise de complexidade. A execução total tem comportamento aproximadamente O(P*logP)

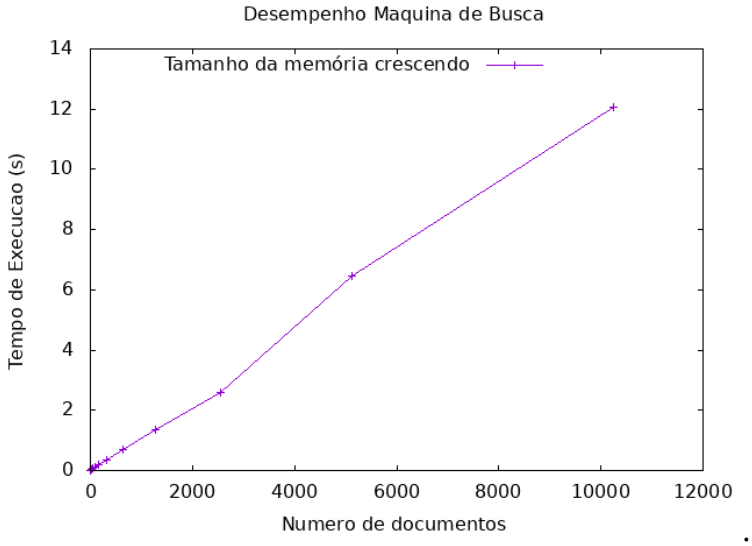
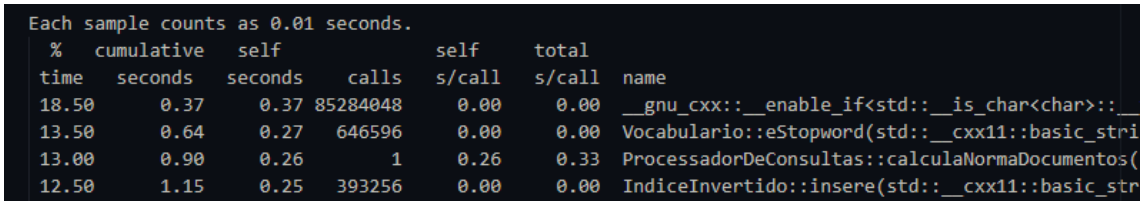


Figura 1: gráfico de desempenho de tempo

Além disso, analisando o perfil de execução do código com o programa GPROF, chega-se a outras conclusões:

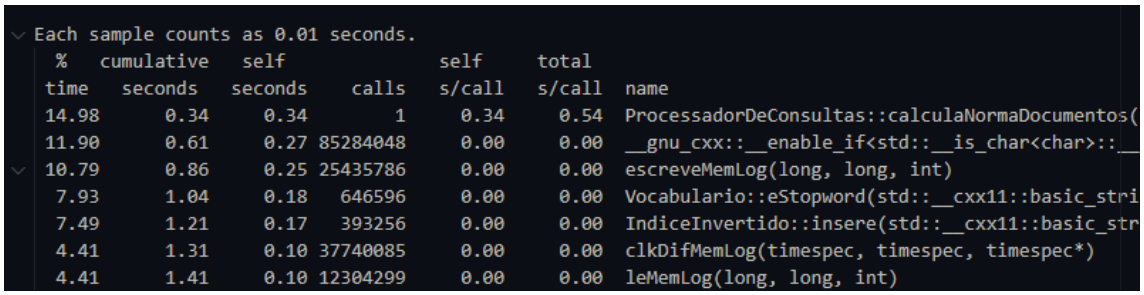
Do tempo de execução total do programa, boa parte pertence aos métodos eStopword, calculaNormaDocumentos e insere.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 18.50      0.37     0.37 85284048     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__
 13.50      0.64     0.27   646596     0.00     0.00  Vocabulario::eStopword(std::__cxx11::basic_stri
 13.00      0.90     0.26        1     0.26     0.33  ProcessadorDeConsultas::calculaNormaDocumentos(
 12.50      1.15     0.25   393256     0.00     0.00  IndiceInvertido::insere(std::__cxx11::basic_str
```

Figura 2: perfil de execução do programa

Além disso, o registro de acesso pode aumentar consideravelmente o custo para executar o código, chegando a tomar quase 10% do tempo de execução total.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 14.98      0.34     0.34        1     0.34     0.54  ProcessadorDeConsultas::calculaNormaDocumentos(
 11.90      0.61     0.27 85284048     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__
 10.79      0.86     0.25 25435786     0.00     0.00  escreveMemLog(long, long, int)
  7.93      1.04     0.18   646596     0.00     0.00  Vocabulario::eStopword(std::__cxx11::basic_stri
  7.49      1.21     0.17   393256     0.00     0.00  IndiceInvertido::insere(std::__cxx11::basic_str
  4.41      1.31     0.10 37740085     0.00     0.00  clkDifMemLog(timespec, timespec, timespec*)
  4.41      1.41     0.10 12304299     0.00     0.00  leMemLog(long, long, int)
```

Figura 3: perfil de execução do programa com o registro de acesso ativado

5.2. Localidade de referência

Outra forma de estudar a performance do programa é fazendo um Mapa de Acesso à Memória, o qual permite enxergar claramente como são feitos os acessos à memória do computador durante a execução das operações, tanto no tempo quanto no espaço.

Esse tipo de observação leva em conta os princípios de Localidade de Referência Espacial e Temporal, que dizem que um acesso ao endereço X no tempo T implica que acessos ao endereço X + dX no tempo T + dT se tornam mais prováveis à medida que dX e dT tendem a zero.

Para facilitar essa análise, as funções registradoras (`leMemLog` e `escreveMemLog`) registram uma fase e um ID em todo registro:

- Um ID identifica a qual classe o registro se refere e é atribuído à classe no momento em que são feitos os registros de acesso, pois no arquivo respectivo de cada classe, as funções registradoras estão com o ID correspondente. O ID 0 corresponde ao `ProcessadorDeDocumentos`, o ID 1 corresponde ao `Vocabulario`, o ID 2 corresponde ao `Quicksort`, o ID 3 corresponde à ListaEncadeadaOcorrencia, o ID 4 corresponde à ListaEncadeadaTermoIndice, o ID 5 corresponde ao ProcessadorDeConsultas, o ID 6 corresponde ao indiceInvertido e o ID 7 à main.
- Uma fase representa um intervalo de tempo no qual as operações realizadas são semelhantes, o que permite separar o registro em fases com diferentes análises. As fases são definidas através da

função `defineFaseMemLog`, dentro de cada operação do código. Fase 0: Processamento do corpus. Fase 1: Criação do Índice. Fase 2: Processamento de consultas.

Esse estudo foi feito a partir da geração de gráficos com tempo no eixo X e endereço de memória no eixo Y, após análises dos registros gerados com o registro de acesso ativo ao realizar a ordenação de 16 entidades com um tamanho de memória igual a 4.

5.2.1. Gráficos de acesso

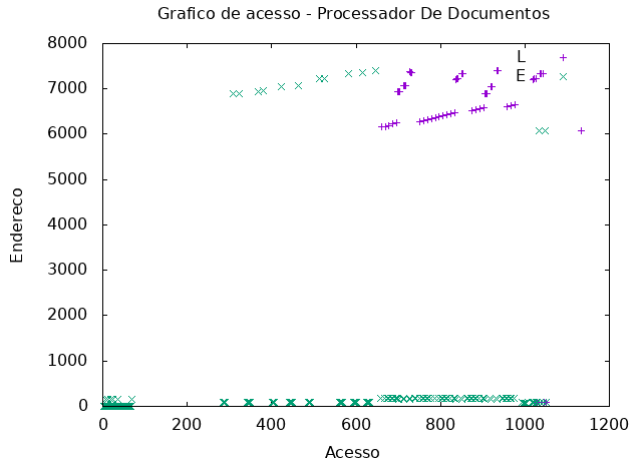


Figura 4: Mapa de acesso à memória da classe Processador De Documentos

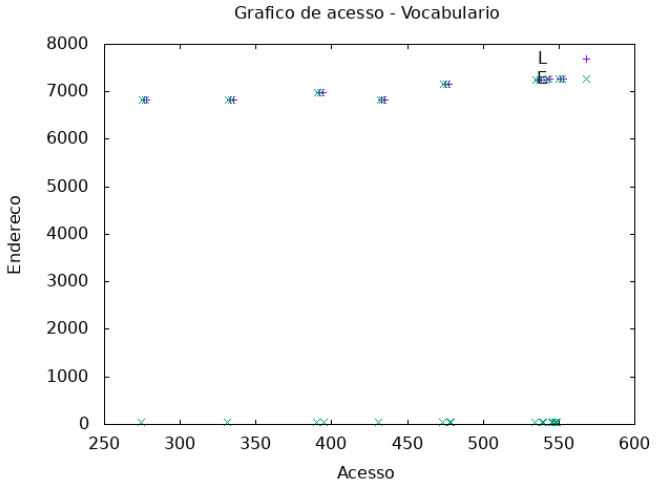


Figura 5: Mapa de acesso à memória da classe Vocabulario

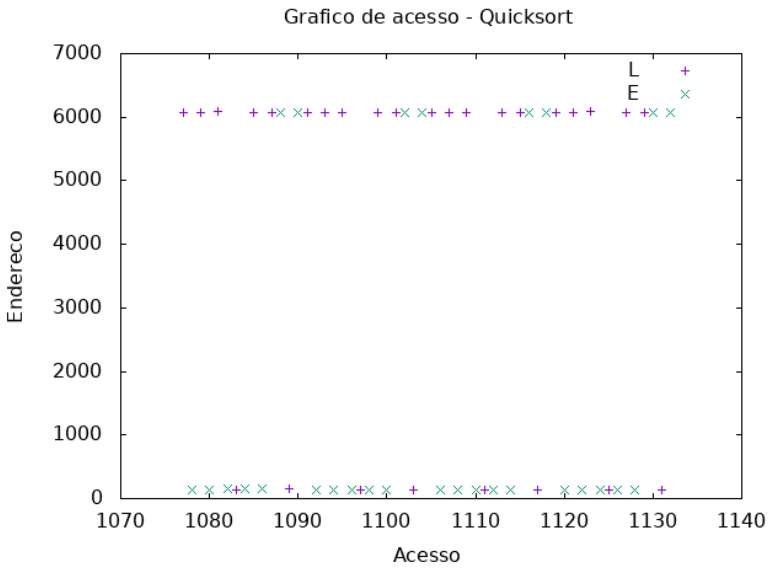


Figura 6: Mapa de acesso à memória da classe Quicksort

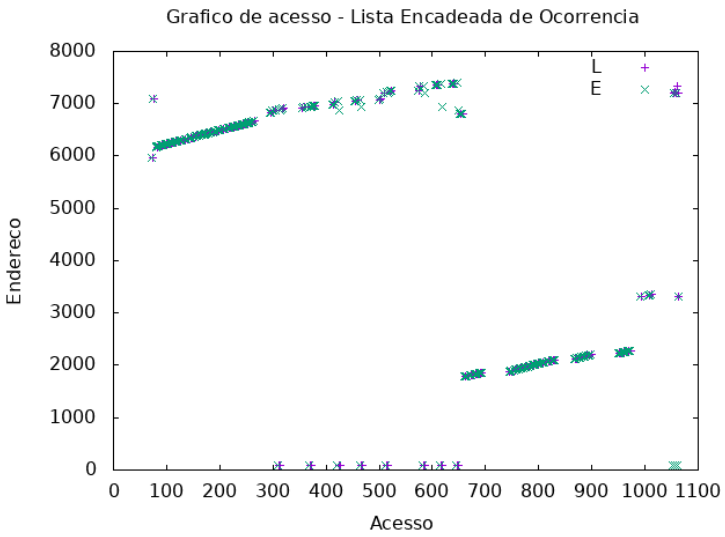


Figura 7: Mapa de acesso à memória da classe Lista Encadeada de Ocorrencia

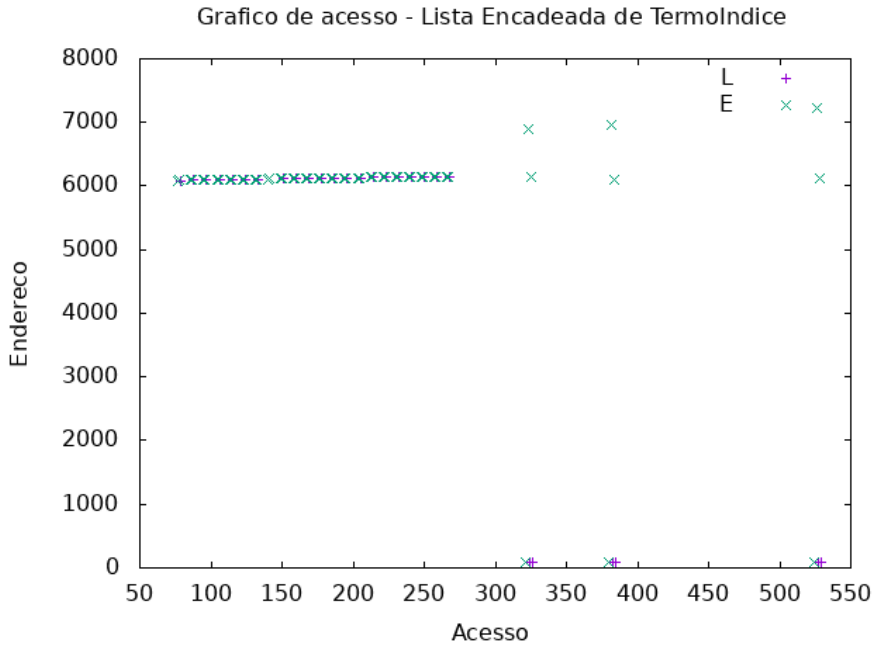


Figura 8: Mapa de acesso à memória da classe Lista Encadeada de Termo Índice

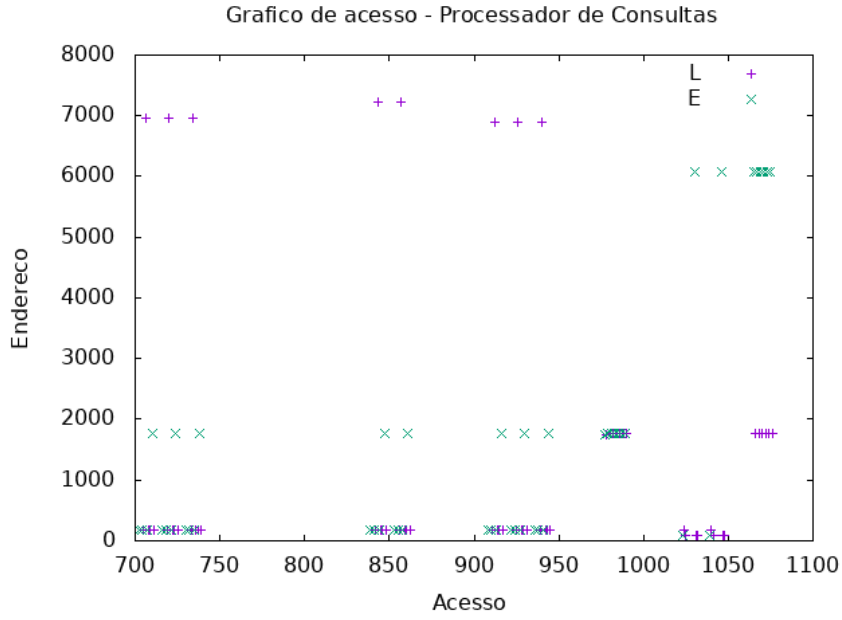


Figura 9: Mapa de acesso à memória da classe Processador de Consultas

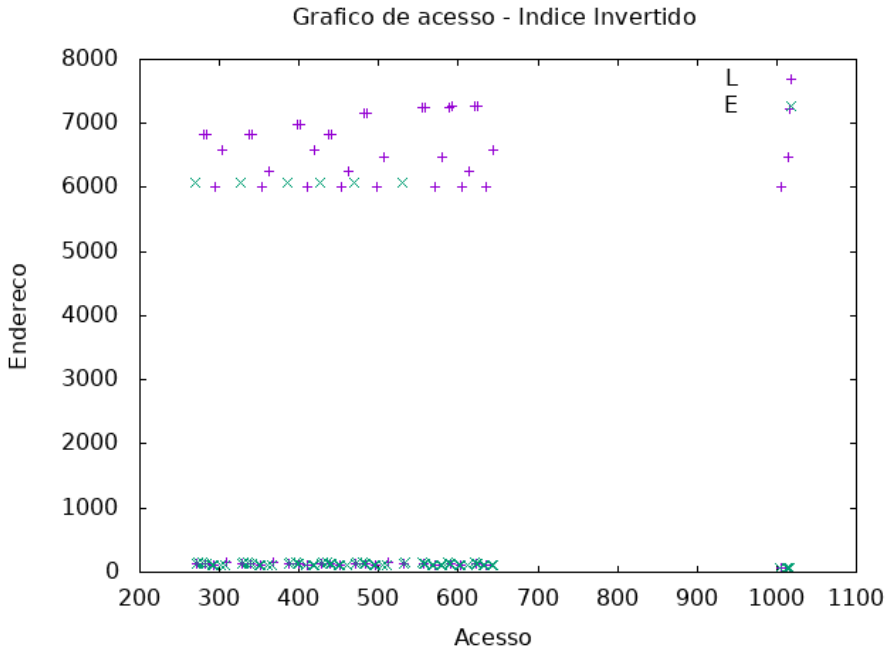


Figura 10: Mapa de acesso à memória da classe Indice Invertido

De posse desses gráficos, podemos ver que a classe com a melhor localidade de referência é a Lista Encadeada de Termo Índice pois todas as outras possuem acessos espalhados pelos endereços. Portanto, para aperfeiçoar o desempenho do programa, seria válido analisar formas de melhorar a localidade de referência dessas classes, evitando movimentações desnecessárias através da memória, mas que não tornassem o código complexo demais. Um exemplo de melhoria possível seria a separação, se possível e de forma razoável, dos métodos que acessam posições afastadas na memória em classes ou etapas diferentes, evitando acessos alternados a localidades distantes.

5.3. Distância de pilha

Outra maneira de analisar a performance do programa e que corrobora com o que já foi analisado anteriormente é a distância de pilha (DP). Cada vez que um endereço é acessado, ele é colocado no topo de uma pilha inicialmente vazia. Quando o endereço é acessado novamente, sua posição na pilha é a distância de pilha.
Fazendo uma soma ponderada de todas as distâncias de pilha com suas frequências de um conjunto de procedimentos, chega-se à distância de pilha desse conjunto e quanto maior ele seja, pior é a localidade de referência da operação. Isso se dá, pois, um mesmo endereço demora mais a ser acessado novamente se sua distância de pilha é alta, ferindo o princípio da localidade de referência. Analisando esses aspectos, é possível tirar algumas conclusões a respeito das diferentes operações e classes.

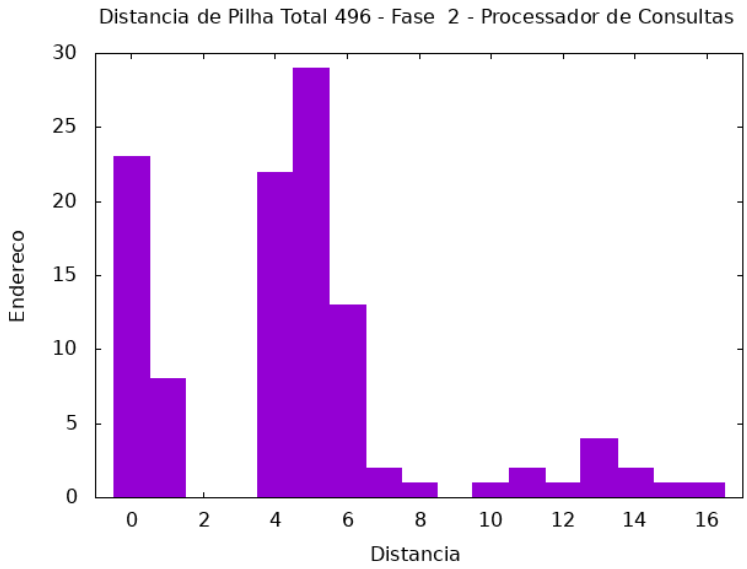


Figura 11: Histograma de distância de pilha da fase 2 da classe Processador de Consultas

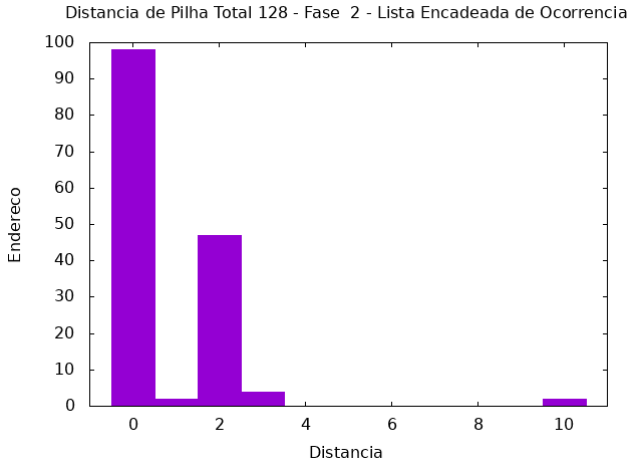


Figura 12: Histograma de distância de pilha da fase 2 da classe Lista Encadeada de Ocorrencia

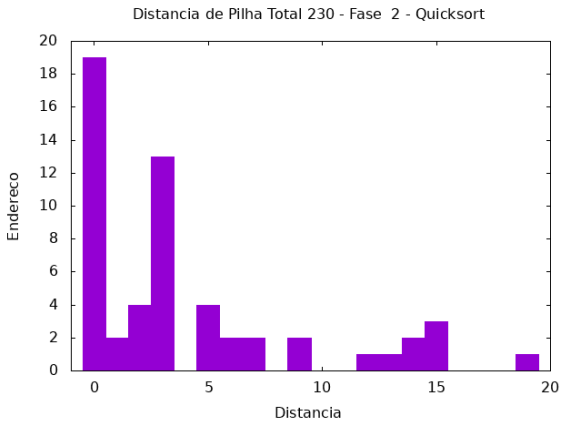


Figura 13: Histograma de distância de pilha da fase 2 da classe Quicksort

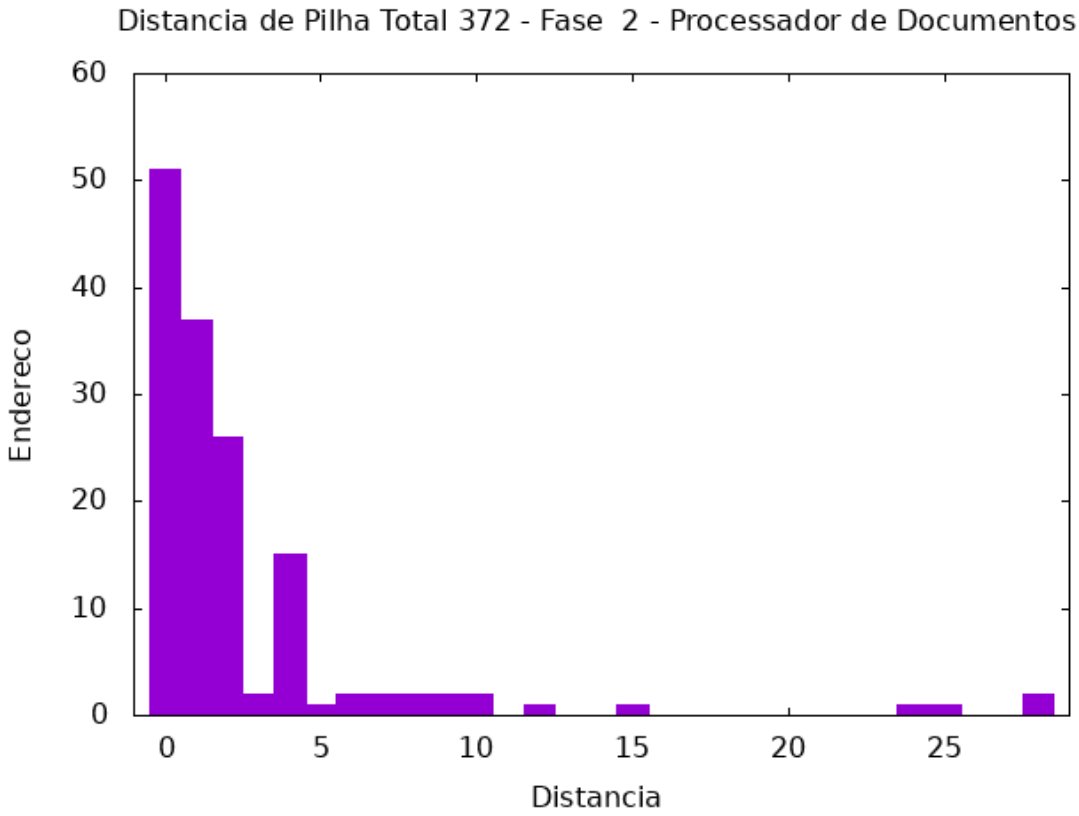


Figura 14: Histograma de distância de pilha da fase 2 da classe Processador de Documentos

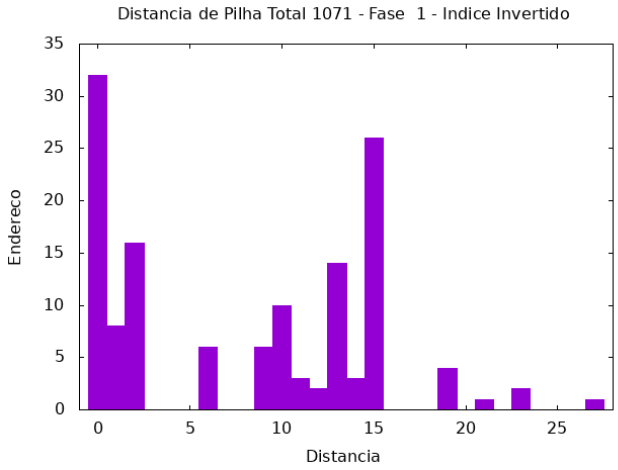


Figura 15: Histograma de distância de pilha da fase 1 da classe Indice Invertido

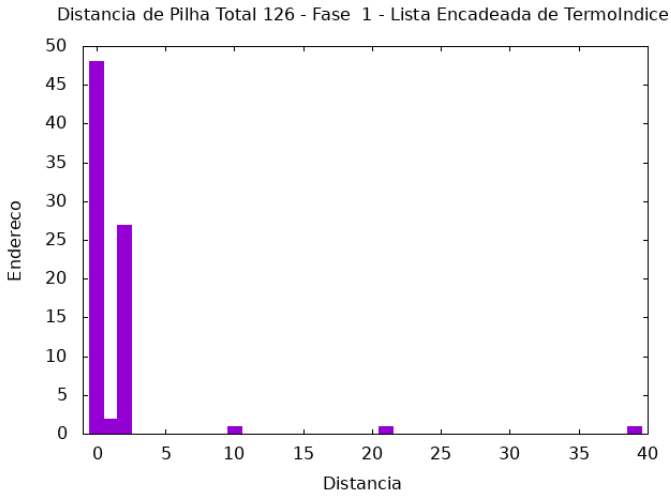


Figura 16: Histograma de distância de pilha da fase 1 da classe Lista Encadeada de Ocorrencia

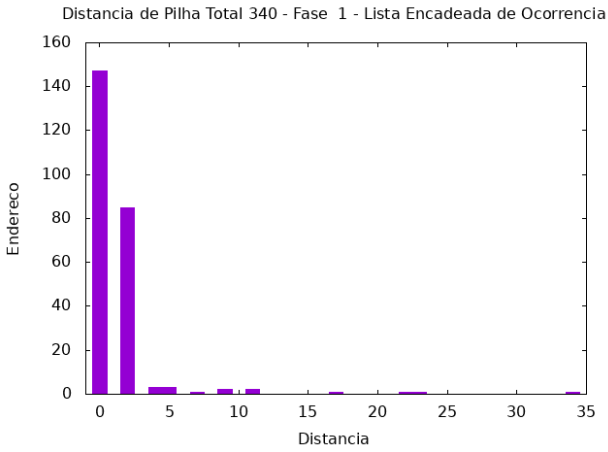


Figura 17: Histograma de distância de pilha da fase 1 da classe Lista Encadeada de Ocorrencia

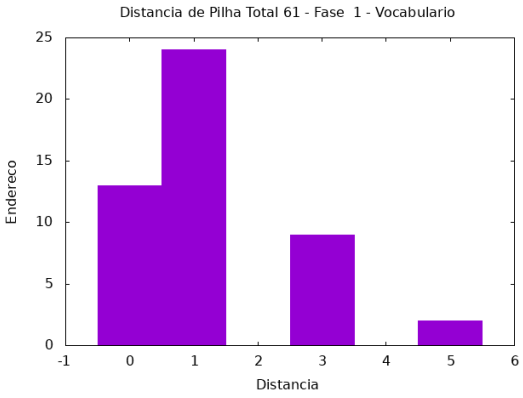


Figura 18: Histograma de distância de pilha da fase 1 da classe Vocabulario

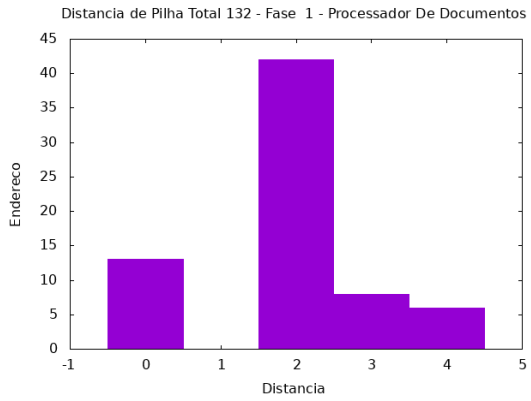


Figura 19: Histograma de distância de pilha da fase 1 da classe Processador De Documentos

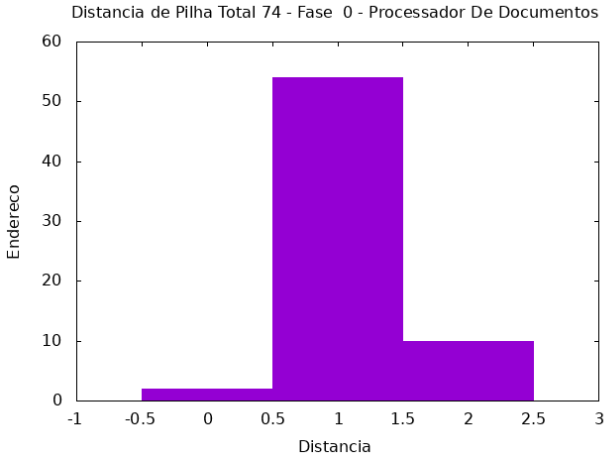


Figura 20: Histograma de distância de pilha da fase 0 da classe Processador De Documentos

As classes e fases com as maiores distâncias de pilha são: fase 1 da classe Indice Invertido, fase 2 da classe Processador de Documentos e fase 2 da classe Processador de Consultas. O índice invertido é bastante requisitado assim como os dois processadores, por isso possuem distâncias de pilha maiores.

6. Conclusão

Este trabalho lidou com a tarefa de implementar uma máquina de buscac, na qual a abordagem utilizada para a resolução foi a implementação de uma tabela de hash. Com a solução adotada, pode-se resolver um problema muito importante e interessante de uma forma eficiente. Por meio da resolução desse trabalho, foi possível aplicar e revisar conceitos relacionados a estruturas de dados, pesquisa, ordenação, alocação de memória, análise de complexidade de tempo e espaço, programação defensiva e performance de programas.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, entender a especificação, implementar o

hash, definição das operações e responsabilidades relevantes a cada classe, análise experimental do desempenho do programa, organização do projeto, erros recorrentes e definição das frentes de trabalho a serem seguidas e a criação de uma documentação completa.

7. Bibliografia

Chaimowicz, L. e Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

https://www.cplusplus.com/reference/cctype/tolower/

https://www.youtube.com/watch?v=j9yL30R6npk&ab_channel=CodeVault

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

https://stackoverflow.com/questions/612097/how-can-i-get-the-list-of-files-in-a-directory-using-c-or-c

Instruções para compilação e Execução

1 – Extraia o arquivo `.zip` na pasta desejada.

2 – Execute o comando "`cd TP`" no terminal

3 – Execute o comando "`make all`" no terminal para compilar os módulos do programa

4 – Execute o programa `main` da pasta `bin` passando um arquivo de texto com os comandos pela linha de comando.