

Trabalho Prático 3: Máquina de Busca Avançada

Mateus Brandão Damasceno Góes - 2020054706

Universidade Federal de Minas Gerais (UFMG)
Sistemas de Informação

1. Introdução

O objetivo desse Trabalho é criar uma máquina de busca dividida em três partes, o crawler que realiza a coleta dos documentos, o indexador que lê os documentos e constrói o índice invertido e o processador de consultas, que consulta o índice e ordena os documentos de acordo com a relevância com a consulta.

Para a construção do índice invertido foi criado Tabelas Hash para a consulta rápida dos elementos, primeiramente os elementos do arquivo de StopWords são armazenados em sua Tabela Hash, Após isso as palavras dos vocabulários são tratadas baseadas na tabela ASCII, remoção de pontuações e transformação em lowercase, é também consultado a presença das palavras na tabela de StopWords, para a devida remoção da palavra do vocabulário, após isso constrói o Hash dos arquivos, para realizar a busca usando o nome do arquivo como chave, dessa forma realizam todas as operações necessárias para o cálculo e consulta dos arquivos, que vão ser mais detalhadas abaixo.

2. Implementação

O programa foi dividido em três partes que serão detalhadas a seguir. Porém inicialmente é importante explicitar algumas classes criadas.

A classe **Palavra** foi criada como forma de servir como elemento inserido na tabela Hash, e possui o atributo da “chave” como variável, porém cada elemento da classe Palavra também possui uma Lista de índices chamada **ListaIndice**, a ListaIndice possui os valores que serão armazenados por palavra, principalmente o arquivo de origem (“arquivo”), a “frequencia” e o valor do Wtd de cada palavra. Observe que como as TabelasHash foram usadas com vários objetivos diferentes, nem sempre será utilizado todas as variáveis da classe palavra, quando isso ocorre as strings são iniciadas com espaços em branco e os números inicializados como 0.

Primeira Parte: Criação do Índice Invertido:

Inicialmente o programa lê os arquivos indicados pela linha de comando e salva em suas respectivas variáveis com nomes explicativos “path + ...”.

Após isso são chamadas funções que servem para contar a quantidade de elementos que cada um dos argumentos possui, no caso, ele conta a quantidade de StopWords presente, a quantidade de Arquivos no Corpus e o tamanho do vocabulário (contando também palavras repetidas). Essa contagem serve para inicializar as três tabelas Hash com seus respectivos tamanhos.

Classe **TabelaHash**: Essa classe serve para armazenar os valores das palavras em suas localizações após aplicada a função Hash().

Para realizar a Função de Hashing, foi pegado a chave (string) e feito uma soma do valor ASCII de cada caracter multiplicado por uma constante prima elevado a posição do caracter, após isso é feito a operação de módulo com o tamanho da tabela:

$$\sum_{k=i}^j s[k] \cdot p^k \mod m$$

Foi determinado essa fórmula usando a constante prima = 31 (esse número foi escolhido pois é superior aos valores que seram passados – caracteres minúsculos de ‘a’ a ‘z’) pois dessa maneira, é minimizado as possibilidades de colisão de strings que possuem os mesmos caracteres, por exemplo: ‘arco’ e ‘orca’ e também a possibilidade de dois valores que possuem apenas um caracter e tamanhos diferentes avaliarem para a mesma chave, exemplo: ‘a’ e ‘aaa’.

A resolução de conflitos da Tabela é feita através de Lista Encadeada, estabelecida na classe **ListaEncadeada**, em que cada Lista Armazena uma **Palavra**

As palavras contidas em StopWords são inseridas na sua tabela Hash, após isso inicia a inserção das palavras do vocabulário. Inicialmente abre arquivo por arquivo do Corpus, usando a biblioteca <dirent.h> . é lido o texto presente no arquivo e removido a pontuação, números, caracteres não-ASCII e é transformado para lowercase usando funções da biblioteca string e bits.

O texto é dividido em palavras, as palavras são buscadas na tabela de stopWords para determinar se devem ou não entrar no vocabulário, se passam, são inseridas no vocabulário, se a palavra já existe, aumenta a frequência dela. Nesse momento também, a Tabela Hash de Arquivos é populada, dessa maneira, nessa tabela, a chave agora passa a ser o nome do arquivo e as palavras são inseridas em seu índice.

Após todos esses passos, foi criado o Índice invertido, que está presente na TabelaHash HashVocabulário, a chave é o termo que é objeto da classe **Palavra** , possuindo como variável, o id do documento de onde veio e a frequência do termo no documento. Essa tabela portanto, armazena o índice Invertido.

Segunda Parte: Calculo do Wt,d e Wd

O cálculo Do Wt,d é feito dentro da classe **ListaIndice** chamado pelas classes TabelaHash e ListaEncadeada.

Esse cálculo é feito utilizando as duas tabelas Hash, a do vocabulário e a dos Arquivos. Primeiramente itera pela tabela de Arquivos, pegando as palavras presente em cada arquivo e chamando de chave, essa chave é procurada na Tabela de vocabulário, o tamanho da ListaEncadeada da PalavraAux obtida é a quantidade de documentos que possuem a chave

Utilizando a fórmula passada, é calculada o $W_{t,d}$ para cada palavra, utilizando como parâmetro também a quantidade de documento do Corpus “quantidadeDocumentos”.

O cálculo do W_d é feito na classe Palavra, chamado através da TabelaHash de arquivos, o cálculo é feito percorrendo cada elemento da lista encadeada de cada arquivo, o $W_{t,d}$ ao quadrado de cada palavra é somado e é feito a raiz quadrada do total, adquirindo um W_d para cada arquivo presente.

Dessa maneira, o $W_{t,d}$ calculado usando a tabela de Vocabulários foi transferido para a Tabela de Arquivos e o W_d permanece na tabela de arquivos, dessa maneira, a tabela de arquivos armazena individualmente o $W_{t,d}$ de cada palavra e o W_d de cada arquivo.

Terceira Parte:Consulta

Para armazenar os resultados foi criado uma classe chamada **ListaConsulta**, que possui seus nodes e cada node possui as variáveis nomeArquivo e resultado, o resultado armazena o total, do somatório dos $W_{t,d}$ das palavras pesquisadas dividido pelo W_d do documento.

A consulta é feita usando a função Consultar da **TabelaHash**, e utiliza a tabela de Vocabulário e de Arquivos, inicialmente procura os documentos em que a palavra está presente, usando a tabela de vocabulários. Os documentos que possuem a palavra são armazenados em uma ListaConsulta.

A partir disso itera por essa lista gerada, procurando os documentos na tabela, para cada palavra encontrada, itera pelos seus elementos “índice”, pegando o $W_{t,d}$ da palavra no documento buscado, faz a divisão desse valor e armazena em resultado.

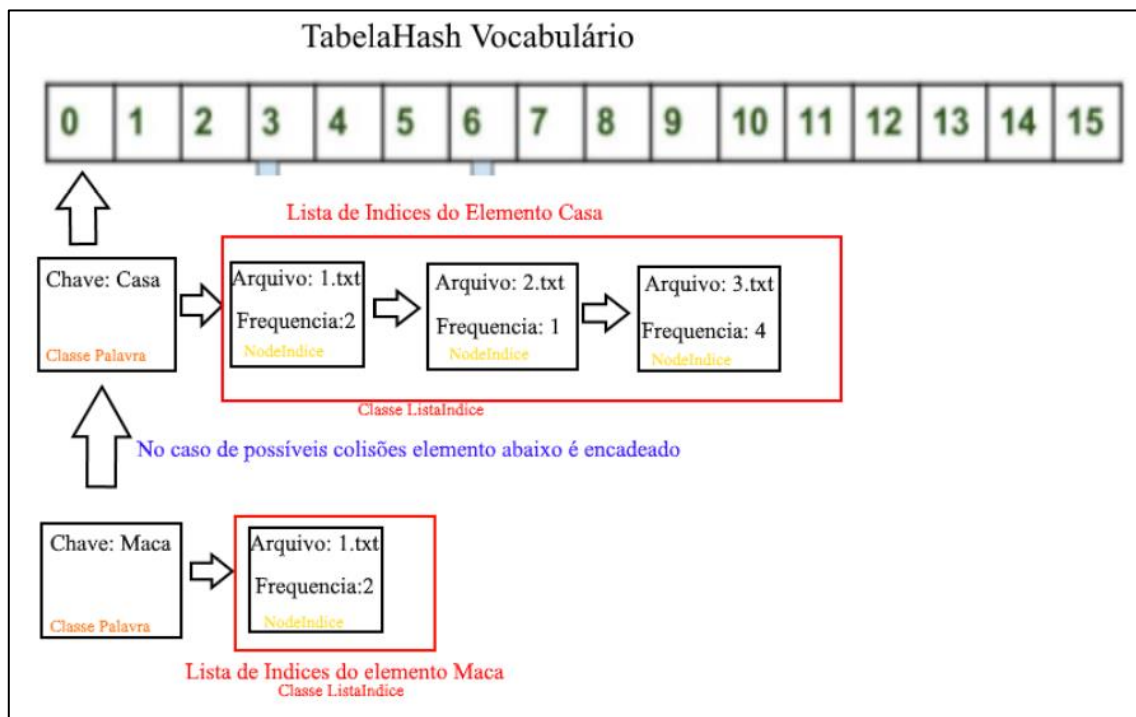
Se a busca possui mais de uma palavra, soma-se o novo valor ao resultado presente na lista, essa operação pode causar leve imprecisão no resultado final pois, é feito da seguinte maneira para pesquisas com mais de uma palavra:

$$\frac{W_{i,q(1)}}{W_d} + \frac{W_{i,q(2)}}{W_d} + \dots \text{diferentemente do sugerido: } \frac{W_{i,q(1)} + W_{i,q(2)}}{W_d}.$$

Ambas as equações geram o mesmo resultado, mas podem gerar leves diferenças devido a maneira como são feitas as operações com floats.

Os nomes e Resultados são armazenados na Lista e é feito MergeSort para determinar qual resultado é maior, no caso de empate, é checado o id do documento, permanecendo o menor id primeiro.

Abaixo é feito um diagrama de visualização da TabelasHash para melhorar o entendimento do programa:



Para a TabelaHash de Arquivos, a chave passa a ser o nome do Arquivo e a variável armazenada em “Arquivo” passa a ser a palavra presente no arquivo, armazena também o Wd e o Wtd de cada palavra

3. Análise de complexidade

Determinar o tamanho: Para determinar a quantidade dos arquivos, quantidade de palavras no vocabulário e quantidade de StopWords, é necessário iterar por todas as palavras ou documentos presentes em cada um desses, logo é $O(n)$ sendo n o número de arquivos, ou o número de palavras presentes.

Inserção na TabelaHash: Devido a função de Hash escolhida visando minimizar os casos onde ocorrem colisões, assume-se que não haja muitos casos em que seja necessário iterar pela lista para inserir, logo como a determinação da posição no array é feita através de operação matemática, a inserção é $O(1)$. Porém o espaço necessário para a construção da tabela é $O(n)$ devido ao armazenamento de N espaços no array da tabela.

Pesquisa na TabelaHash: Para a pesquisa, é necessário iterar pelos elementos da Lista de colisões presente na posição determinada pelo Hash, normalmente, assumindo que não haja colisões, não é necessário iterar pela lista, portanto a pesquisa é $O(1)$, no caso da necessidade de iterar, a pesquisa se torna $O(k)$ sendo k a quantidade de elementos na ListaEncadeada de colisões em determinada chave.

Cálculo do Wt,d: O cálculo do Wt,d como utiliza consultas na Tabela Hash de vocabulários, que tem caso médio $O(1)$, porém é necessário iterar por cada elemento da Tabela Hash de Arquivos, ou seja, dessa maneira fazendo um cálculo para cada palavra presente em cada documento. Ou seja, sendo N o número de palavras, M o

número de documentos e K o número de palavras por documento, são feitas N consultas usando $O(1) \rightarrow O(N)$ + são feitas K consultas para cada M documentos $\rightarrow O(M \times K)$, por fim resultando em $O(N) + O(M \times K)$.

Cálculo do Wd: O cálculo do Wd é feito um por arquivo, baseado em quantas palavras tem em cada arquivo, seja M arquivos com N palavras cada, temos $O(N \times M)$.

Ordenação da Lista de Resultados: A ordenação na Lista de Resultados foi feita utilizando MergeSort, é feita a divisão da lista em até $\log N$ partes, e o processo de merge itera por toda a lista tomando $O(N)$, logo a complexidade de tempo se dá por $O(N) \times O(\log N) = O(N \times \log N)$ e o espaço tomado é dado por $O(N)$ visto que ocorre o armazenamento completo da lista de Resultados.

Essas foram as análises de complexidade dos principais passos do meu programa. O programa no geral é bem rápido visto a quantidade de palavras que precisam ser tratadas, isso se deve a utilização de Tabelas Hash durante todos os processos do programa, agilizando o processo de consulta e inserção nas tabelas. Dessa maneira, a maioria dos subprocessos feitos pelo programa se tornou $O(1)$ ou $O(N)$ dependendo dos dados tratados.

4. Estratégias de Robustez

As principais estratégias para evitar erros relacionados a memória se dão na iteração através das listas, que ocorrem sempre checagem do elemento atual, se está contido na lista, para evitar o acesso indevido de outras localizações de memória. Além disso a criação do destrutor da classe TabelaHash, que deleta os elementos alocados por ela, liberando mais espaço de memória.

Outro problema que pode ser evitado seria a impossibilidade da abertura de um diretório no início da leitura dos dados, se a pasta não pode ser acessada ocorre a impressão de um erro para alertar o usuário.

5. Testes

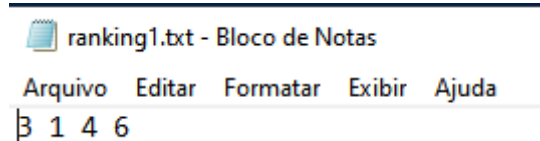
Foram realizados vários testes de acordo com as consultas-teste enviadas. Primeiramente realizei um teste baseado na Descrição do Trabalho Prático, criei 6 arquivos simples igual os que foram descritos, contendo as palavras casa, lua e maca. Fiz dessa maneira para testar passo a passo do programa e ir conferindo os resultados, já que cada etapa foi disponibilizada na descrição.

Primeiro Teste

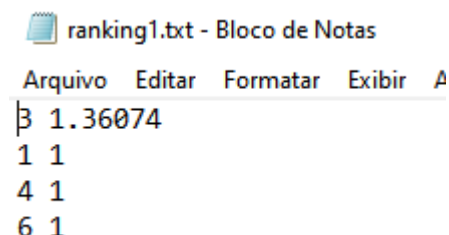
Usando as palavras passadas, e as stop words “a”, “da” e “e”. Ao realizar a busca “casa maca” recebemos o resultado esperado $\rightarrow 3,1,4,6$

```
C:\Users\Mateus\Desktop\TP03\bin>Busca.exe -i ../consulta1.txt -o ../ranking1.txt -c ../colecão_teste -s ../stopwordsteste.txt
```

Com os arquivos presentes na raiz do diretório “TP03” foi passado o caminho como busca relativa, produzindo o arquivo “ranking1.txt”



Para demonstrar que após todos os cálculos o programa produz os mesmos resultados esperados para $\text{Sim}(d,q)$ modifiquei o programa para que seja impresso os resultados juntamente com a ordem dos arquivos, o resultado da busca foi a seguinte:



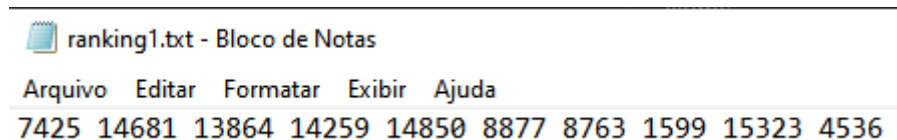
Dessa maneira pude notar que os valores calculados correspondem ao resultado esperado demonstrado na descrição do Trabalho Prático.

Segundo Teste

Foi feito um teste também com a `colecão_small` disponibilizada para conferir se os arquivos batem com o ranking gerado. Dessa vez a palavra utilizada na consulta foi “laptop”

```
C:\Users\Mateus\Desktop\TP03\bin>Busca.exe -i ../consulta1.txt -o ../ranking1.txt -c ../colecão_small -s ../stopwords.txt
```

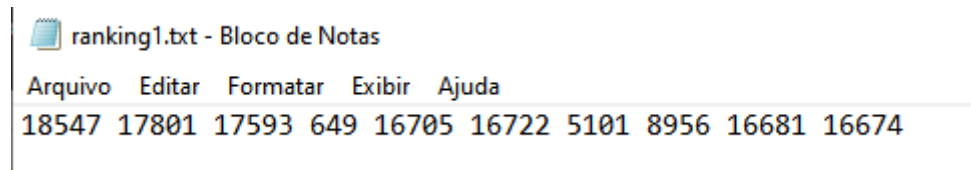
Produzindo o seguinte Ranking:



O ranking difere levemente do ranking incluído como resposta pelos professores, isso provavelmente se deve a maneira como foi feita o cálculo e pela leve diferença entre a quantidade dos arquivos, porém percebe-se que os mesmos arquivos compõem o ranking, apenas algumas das ordens são diferentes.

Terceiro Teste

O terceiro teste foi feito usando a colecao_full com 18637 arquivos, utilizando como consulta a query “usb cable”. Produzindo o seguinte resultado:



18547	17801	17593	649	16705	16722	5101	8956	16681	16674
-------	-------	-------	-----	-------	-------	------	------	-------	-------

Novamente com variações das posições do ranking porém mantendo os elementos esperados presentes no arquivo

6. Conclusão

Esse trabalho foi bem desafiador devido ao fato de envolver várias áreas das quais foram estudadas ao longo do semestre, acredito que serviu bem como maneira de aprendizado. Tive dificuldades na visualização da maneira em como os dados estavam organizados na Tabela inicialmente, devido ao fato da utilização de várias listas aninhadas e diversas estruturas de dados em conjunto. Dessa forma, acredito que foi muito útil devido a revisão, estudo e uso prático dos Algoritmos de Pesquisa, Ordenação, Análise de complexidade, etc.

7. Bibliografia

Slides virtuais da disciplina de Estruturas de Dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Fórum de dúvidas da disciplina de Estruturas de Dados. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Mead's Guide to Getopt, Disponível em:

< <https://azrael.digipen.edu/~mmead/www/Courses/CS180/getopt.html> >

Dirent.h File, Disponível em:

< <https://www.ibm.com/docs/en/aix/7.2?topic=files-direnth-file> >

Apêndice – Instruções para compilação e execução

O programa foi testado para sua compilação e Execução no Sistema Operacional **Windows 10**. Para compilar, basta acessar a pasta através do Prompt de Comando e rodar o comando '**make**'. A partir dele, gera o executável na pasta bin.

O programa leva exatamente **4** argumentos para sua execução, o primeiro é o caminho para o documento de consulta, o segundo é o nome do arquivo que vai ser gerado, o terceiro argumento é o caminho para o corpus, o quarto é o caminho para as stopwords.

OBS : Os testes realizados por mim utilizaram caminho relativo, ou seja, o executável se encontrava na pasta bin e os arquivos na pasta raíz, superior ao diretório bin, acessando a pasta raíz utilizando “../”

Logo, no cmd pode ser executado da seguinte maneira:

**Busca.exe -i <Caminho do arquivo de consulta> -o <Arquivo de output >
-c <Caminho para o corpus> -s <caminho stopwords>**

Um exemplo de como eu realizei os testes foi da seguinte maneira:

Busca.exe -i ../consulta1.txt -o ../ranking1.txt -c ../colecão_small -s ../stopwords.txt

Os arquivos todos estavam presentes na pasta TP03 e o executável na pasta TP03/bin

Detalhe para a importância de não colocar a barra ao final do argumento do diretório que compõe o corpus.

Acredito que o programa funcione como o esperado apenas no sistema Windows 10 devido a necessidade de acesso aos diretórios, que tem implementação específica para o sistema operacional.