Trabalho Prático 3 em Estruturas de Dados

Máquina de Busca

Gustavo Freitas Cunha

Matrícula: 2020054498

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1. Introdução

Nos últimos anos, a web tem passado por uma revolução na geração de dados. O grande desafio advindo desse crescimento, é a recuperação de informação. Para que se obtenha uma informação disponível na web atualmente, normalmente é utilizada uma máquina de busca, um mecanismo complexo que demanda robustez e muita engenharia de software de quem o projeta. Neste trabalho, serão implementados os dois principais componentes de uma máquina de busca: o **indexador de memória** que, de acordo com o **vocabulário** e os documentos que compõem o **corpus** da web, cria um **índice invertido**, mapeando os termos e documentos e um **processador de consulta**, que ordena os documentos recuperados de acordo com sua similaridade com a consulta.

2. Implementação

O programa foi desenvolvido e testado em sistema operacional Subsistema Windows para Linux (Windows Subsystem for Linux - WSL), na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection, em processador Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz, com 8GB de RAM instalada.

2.1 Organização do Código

Este projeto foi estruturado da seguinte forma:

```
- TP
  |- src
  |- bin
  |- obj
  |- include
  Makefile
```

O diretório `src` armazena os arquivos de código (extensão `.cpp`), o diretório `include` os arquivos de cabeçalho (extensão `.h`), o diretório `obj` armazena os

arquivos objects (extensão `.o`) e o diretório `bin` armazena os executáveis (`.exe`). Os diretórios `obj` e `bin` foram entregues vazios, conforme instrução, mas após compilação e linkagem pelo Makefile, os arquivos gerados são alocados em suas respectivas pastas.

2.2 Estrutura de Dados

Dentre as estruturas de dados modeladas para este trabalho, estão:

- **class Celula**: classe que possui um item do tipo T **(template)** e um ponteiro para Celula, chamado prox. A clássica Celula usada em ListaEncadeada.

- **class ListaEncadeada**: A clássica classe de Celulas encadeadas com ponteiros para a primeira e a última células, além de um atributo tamanho. Neste trabalho, essa classe foi implementada para ser um lista de elementos da classe T **(template)**.

- **class Documento:** esta classe representa um documento do corpus e possui atributos id, Wd, similaridade e uma ListaEncadeada<std::string>, denominada vocabularioLocal que, como o próprio nome diz, encadeia as palavras que aquele documento possui.

- **class IndiceInvertido:** esta classe representa o par ordenado definido na especificação do projeto:

  (id do documento, frequência do termo no documento)

- **class Termo:** esta classe representa as palavras. Possui a string termo, que é a palavra propriamente dita e uma ListaEncadeada<IndiceInvertido>, conforme definido na especificação do projeto.

- **class Hash:** esta classe representa a tabela hash solicitada. É um vetor tamanho **M** (número de termos do vocabulário geral do corpus) de ListaEncadeada<Termo>. A classe se encarrega de armazenar, através de sua função **hashing**[1] os termos trabalhados e os índices invertidos respectivos, **tratando colisões com encadeamento de termos.**

[1]Utilizada a função indicada: https://cp-algorithms.com/string/string-hashing.html

2.3 Funcionamento do Programa

A função main deve receber, **obrigatoriamente**, os 4 argumentos a seguir:

1) Caminho/nome do arquivo .txt referente à consulta, com a **flag -i**;
2) Caminho/nome do arquivo .txt de saída, (contém índices dos documentos do corpus ordenados de acordo com sua relevância), com a **flag -o**;
3) Caminho/nome da pasta onde estão os documentos que compõe o corpus, com a **flag -c;**
4) Caminho/nome do arquivo .txt que contém as stop words, com **flag -s.**

O programa principal é main.cpp. É nele que está a função main e é feita a leitura e processamento dos arquivos de entrada passados na linha de comando, bem como a chamada das funções de acordo com cada etapa, definidas a seguir:

ETAPA 1: Processamento dos parâmetros recebidos, por meio da função getopt.

ETAPA 2: Processamento das stopwords, de acordo com o documento correspondente recebido como parâmetro.

ETAPA 3: Processamento do tamanho do vocabulário e tamanho do corpus, percorrendo todos os documentos que o compõem, com a função contaCorpus e filtrando tudo com a função filtraLinha, para remoção de caracteres indesejados.

ETAPA 4: Construção da tabela Hash, por meio da função constroiTabelaHash (filtrando tudo com a função filtra linha, para remoção de caracteres indesejados), onde também é construída uma ListaEncadeada<Documento>, como a setagem, em cada elemento da lista, do respectivo id e encadeamento, na respectiva ListaEncadeada<std::string> vocabularioLocal, das palavras correspondentes.

ETAPA 5: Processamento da consulta, de acordo com o documento correspondente recebido como parâmetro, com a função processaPalavrasConsulta, também utilizando a mesma filtragem da função filtraLinha.

ETAPA 6: Cálculo das similaridades de cada documento com a consulta solicitada, com a função calculaSimilaridade.

ETAPA 7: Ordenação e rankeamento no arquivo de saída solicitado, dos documentos de acordo com a similaridade calculada, com a função ordena e sort.

2.4 Observações importantes

- **Este programa apenas funciona caso todos os parâmetros de entrada OBRIGATÓRIOS, conforme especificado na seção 2.3, tenham sido passados na linha de comando.**

3. Análise de Complexidade

As variáveis representadas nas notações assintóticas estão descritas na respectiva coluna de comentários.
Complexidade de funções de bibliotecas importadas foram consideradas constantes, conforme orientação.

class ListaEncadeada				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas inicializa ponteiros - O(1).	O(1)	Alocação dinâmica de uma variável do tipo Celula, apenas.
Posiciona	Melhor Caso: O(1) Pior caso: O(n)	Caso a lista esteja vazia, tenha apenas um elemento ou a posição solicitada seja a primeira, a complexidade é constante, caso contrário, a complexidade é linear, conforme tamanho da lista (n).	O(1)	Apenas declaração de ponteiros e variáveis auxiliares.
GetItem	Melhor Caso: O(1) Pior caso: O(n)	Chama Posiciona. As outras operações têm complexidade constante.	O(1)	Apenas declaração de ponteiros - O(1) - e chamada para Posiciona, que é O(1).
GetTamanho e GetPrimeiro	O(1)	Apenas retorna atributos da classe - O(1).	O(1)	Não há alocações ou declarações.
ObtemElemento	Melhor Caso: O(1) Pior caso: O(n)	Caso a lista esteja vazia, tenha apenas um elemento ou caso a URL pesquisada esteja na primeira posição, a complexidade é constante, caso contrário, a complexidade é linear, conforme tamanho da lista (n).		Apenas declaração de um ponteiro.
InsereFinal	O(1)	Apenas manipulação de ponteiros - O(1).	O(1)	Declaração e alocação de 1 ponteiro, apenas.

Contem	Melhor Caso: O(1) Pior caso: O(n)	Caso a lista esteja vazia, tenha apenas um elemento ou caso a URL pesquisada esteja na primeira posição, a complexidade é constante, caso contrário, a complexidade é linear, conforme tamanho da lista (n).	O(1)	Apenas declaração de um ponteiro.
Imprime	O(n)	Imprime todos os elementos da lista em um laço que navega por todas as suas posições. Logo, complexidade varia linearmente conforme tamanho da lista (n).	O(1)	Apenas declaração de um ponteiro.
Limpa	O(n)	Navega pelas células da lista, excluindo-as. Por isso, a complexidade varia linearmente conforme tamanho da lista (n).	O(1)	Apenas declaração de um ponteiro.
Destrutor	O(n)	Chama Limpa - O(n).	O(1)	Chama Limpa, que é O(1).

Tabela 1 - Complexidade de tempo e espaço dos métodos da classe ListaEncadeada

class Documento				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas alocações e setagem de atributos - O(1).	O(1)	Alocação dinâmica de uma ListaEncadeada<std::string>, usando o respectivo construtor - O(1) - e setagem de atributos - O(1).
Getters e Setters	O(1)	Apenas retorno e setagem de atributos da classe - O(1).	O(1)	Não há declarações ou alocações.

Tabela 2 - Complexidade de tempo e espaço dos métodos da classe Documento

class IndiceInvertido				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas alocações e setagem de atributos - O(1).	O(1)	Não há declarações ou alocações
Getters e Setters	O(1)	Apenas retorno e setagem de atributos da classe - O(1).	O(1)	Não há declarações ou alocações.
IncrementaFrequencia	O(1)	Como o próprio nome diz, apenas incrementa o atributo frequência - O(1).	O(1)	Não há declarações ou alocações.
operator==	O(1)	Sobrecarga do operador relacional '=='. A igualdade da classe é definida pela igualdade do atributo idDocumento (int). Logo, é feita a verificação dessa igualdade - O(1).	O(1)	Não há declarações ou alocações.
Imprime	O(1)	Imprime, como um par ordenado, os atributos idDocumento e frequencia - O(1).	O(1)	Não há declarações ou alocações.

Tabela 3 - Complexidade de tempo e espaço dos métodos da classe IndiceInvertido

class Termo				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas alocações e setagem de atributos - O(1).	O(1)	Alocação dinâmica de uma ListaEncadeada<IndiceInvertido>, usando o respectivo construtor - O(1) - e setagem de atributos - O(1).
Getters e Setters	O(1)	Apenas retorno e setagem de atributos da classe - O(1).	O(1)	Não há declarações ou alocações.

operator==	O(1)	Sobrecarga do operador relacional '=='. A igualdade da classe é definida pela igualdade do atributo termo (std::string). Logo, é feita a verificação dessa igualdade - O(1).	O(1)	Não há declarações ou alocações.
Imprime	O(n)	Seja n o tamanho de ListaEncadeada<IndiceInvertido> indices, temos a impressão do atributo termo - O(1) - e a impressão de cada um dos n IndiceInvertido em indices - O(n). Logo, temos O(1) + O(n) = O(n).	O(1)	Não há declarações ou alocações.

Tabela 4 - Complexidade de tempo e espaço dos métodos da classe Termo

class Hash				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas alocações e setagem de atributos - O(1).	O(M)	Seja M o parâmetro recebido (correspondente ao tamanho do vocabulário do corpus). Temos a alocação dinâmica de um vetor de M ListaEncadeada<Termo>, usando o respectivo construtor - M*O(1) = O(M) - e setagem de atributos - O(1). Assim, temos complexidade de espaço O(M).
hashing	O(k)	Seja k o tamanho de palavra (std::string recebida como parâmetro). Temos um laço linear conforme k, executando operações matemáticas constantes. Assim, a complexidade é linear conforme k.	O(1)	Apenas declarações de variáveis auxiliares.
Pesquisa	Melhor Caso: O(k) Pior caso: O(max(k,n))	Seja k o tamanho de palavra (std::string recebida como parâmetro). Chama hashing - O(k) - para identificar posição em tabela. Chama Contem para a ListaEncadeada da posição mapeada (seja n seu tamanho) - Melhor Caso O(1) e Pior Caso O(n).	O(1)	Apenas declarações de variáveis auxiliares.

ObtemTermo	Melhor Caso: O(k) Pior caso: O(max(k,n))	Seja k o tamanho do atributo termo (std::string) do Termo recebido como parâmetro. Chama hashing - O(k) - para identificar posição em tabela. Chama ObtemElemento - Melhor Caso: O(1) Pior caso: O(n) - para a ListaEncadeada (de tamanho n) da posição mapeada..	O(1)	Apenas declarações de variáveis auxiliares.
Insere	O(k)	Seja k o tamanho do atributo termo (std::string) do Termo recebido como parâmetro. Chama hashing - O(k) - para identificar posição em tabela. Chama InsereFinal- O(1) - para a ListaEncadeada (de tamanho n) da posição mapeada. Logo, O(k).	O(1)	Apenas declarações de variáveis auxiliares.
calcula_w_td	Melhor Caso: O(k) Pior caso: O(max(k,n))	Temos várias chamadas para getters e setters (todas O(1)), uma chamada para Hash::ObtemTermo, com atributo termo de tamanho k e uma chamada para ListaEncadeada<IndiceInvertido>::ObtemElemento de tamanho n, além de operações matemáticas de custo constante.	O(1)	Apenas declarações de variáveis auxiliares.
calcula_Wd	Melhor Caso: O(nk) Pior caso: O(n*max(k,n))	Temos várias chamadas para getters e setters (todas O(1)). Seja n o tamanho da ListaEncadeada<std::string> vocabularioLocal do Documento d recebido como parâmetro. Temos, para cada posição desta lista, uma chamada para Hash::calcula_w_td, além de operações matemáticas de custo constante.	O(1)	Apenas declarações de variáveis auxiliares.

Tabela 5 - Complexidade de tempo e espaço dos métodos da classe Hash

Programa Principal (main.cpp)				
Método	**Tempo**	**Comentário**	**Espaço**	**Comentário**
filtraLinha	O(k)	Seja k o tamanho da string recebida como parâmetro. Neste método, temos 4 loops que percorrem toda a string (complexidade linear em k) removendo caracteres indesejados definidos na especificação do projeto. Assim, sua complexidade, no geral, é linear conforme k.	O(1)	Apenas declarações de variáveis auxiliares.

contaCorpus	O(Dk)	Seja D o número de documentos do corpus, temos um laço linear em D, que chama filtraLinha - O(k) - e algumas outras operações constantes, como inserção das palavras processadas em ListaEncadeada<std::string>. Logo, temos O(Dk).	O(1)	Apenas declarações de variáveis auxiliares - O(1) - e alocação de uma ListaEncadeada<std::string> - O(1).
adicionaOcorrencia	Melhor Caso: O(k) Pior caso: O(max(k,n))	Primeiramente, temos uma chamada para Hash::Pesquisa - Melhor Caso: O(k) Pior caso: O(max(k,n)), com k sendo o tamanho de termo - Depois, caso termo esteja no hash, temos uma chamada para Hash::ObtemTermo, uma chamada para ListaEncadeada<std::string>::Contem e uma chamada para ListaEncadeada<std::string>::ObtemElemento, além de chamadas para métodos constantes. Caso termo não esteja no hash, temos apenas chamadas para métodos constantes.	O(1)	Apenas declarações e alocações de variáveis auxiliares.
constroiTabelaHash	Melhor Caso: O(Dk) Pior caso: O(D*max(k,n))	Seja D o número de documentos do corpus, temos um laço linear em D, que chama filtraLinha, - O(k) - algumas outras operações constantes, como inserção das palavras processadas em ListaEncadeada<std::string> e uma chamada para adicionaOcorrencia.	O(1)	Apenas declarações de variáveis auxiliares.
processaPalavrasConsulta	O(max(k,m))	Temos um laço que itera sobre as linhas do arquivo recebido como parâmetro, mas esses arquivos têm apenas uma linha. Assim, esse laço é iterado apenas uma vez. Dentro dele, temos uma chamada para filtraLinha, e um laço que itera sobre todas as palavras da linha com operações constantes. Seja m o número de palavras da linha.	O(1)	Apenas declarações de variáveis auxiliares.
calculaSimilaridade	Melhor Caso: O(mk) Pior caso: O(m*max(k,n))	Além dos métodos getters e setters de complexidade constante, temos um laço que navega por todas as palavras da ListaEncadeada<std::string> vocabularioLocal do Documento recebido como parâmetro (seja m o tamanho dessa lista). Dentro desse laço temos, além dos métodos constantes, chamadas para ListaEncadeada<std::string>::Contem e Hash::calcula_w_td.	O(1)	Apenas declarações de variáveis auxiliares.

ordena	O(1)	Apenas verificações de condições, com chamadas de métodos getters constantes e retorno de valores booleanos.	O(1)	Não há declarações ou alocações.

Tabela 6 - Complexidade de tempo e espaço dos métodos do programa principal

4. Estratégias de Robustez

Como estratégias de robustez foram utilizadas as macros definidas em `mgassert.h`. Dentre as macro definidas, temos:

- `avisoAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__avisoAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__avisoAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e linha em que a falha foi detectada. Exibe a mensagem e aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa ficaria comprometida.
- `erroAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__erroAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__erroAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e linha em que a falha foi detectada. Exibe a mensagem, mas não aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa segue normalmente.

Tais mecanismos foram implementados com o fito de tornar o programa mais interativo, dinâmico e robusto, além de facilitar o reconhecimento de falhas das mais diversas naturezas: alocação de memória, abertura de arquivo, verificação de entradas válidas, entre outros.

5. Testes

A fim de verificar o funcionamento da implementação deste pequeno sistema, foram realizados alguns testes.

- **Teste 1**

O primeiro e mais básico teste foi com o corpus e as consultas disponibilizados no enunciado do trabalho, descritos a seguir.

Id do Doc	Conteúdo
1	casa a casa da casa
2	a lua
3	casa casa maca maca maca maca maca lua
4	a maca
5	lua e lua
6	maca

Tabela 7 - Corpus do primeiro teste realizado, com 6 arquivos

O arquivo com as stop words apenas continha as palavras “a”, “da” e “e”.

A primeira consulta foi "casa maca", e o resultado obtido foi, conforme esperado, a impressão no arquivo de saída solicitado: “3, 1, 4, 6”.

A segunda consulta realizada foi "maca lua", também com a impressão do resultado esperado: “ 2, 4, 5, 6, 3”.

A terceira e última consulta para este corpus foi "casa", com saída impressa “1,3” no arquivo solicitado.

- **Teste 2**

O segundo teste realizado foi com o corpus “colecao_full”, com 18637 arquivos, disponibilizado no Moodle e com o arquivo de stop words do Apêndice B.

A primeira consulta testada foi “laptop”, com saída impressa “7425 13864 14681 1599 10486 14850 14259 15323 15529 16017 ”.

A segunda consulta testada foi “android”, com saída impressa “3629 5174 13682 12804 10777 11366 15636 12509 12592 10002”.

A terceira e última consulta testada foi “iridium”, com saída impressa “16281”.

6. Conclusão

Este trabalho teve como objetivo a implementação de dois componentes de uma máquina de busca: um indexador de memória e um processador de consultas. Os principais desafios que enfrentei nesta implementação foram a integração da obtenção dos dados gerados pelo indexador com a ordenação do processador, além da leitura de documentos do corpus, na pasta recebida como parâmetro, já que eu nunca havia trabalhado com iteração em pastas em um projeto. Por fim, acho que o resultado foi

satisfatório e a prática da codificação sempre me ensina muito, além do conhecimento e aprofundamento das estruturas utilizadas, como hash, lista encadeada e outras.

7. Bibliografia

CORMEN , T., LEISERSON, C, RIVEST R., STEIN, C. **Introduction to Algorithms**, Third Edition, MIT Press, 2009.

CHAIMOWICZ, L. and PRATES, R. **Slides virtuais da disciplina de estruturas de dados**, 2020.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Class std::string. **C Plus Plus**, 2000.

Disponível em: <https://www.cplusplus.com/reference/string/string/>.

Acesso em: 11 de fevereiro de 2022.

Class std::vector. **C Plus Plus**, 2000.

Disponível em: <https://www.cplusplus.com/reference/vector/vector/>.

Acesso em: 11 de fevereiro de 2022.

Navegação no sistema de arquivos. **Microsoft Docs**, 2021.

Disponível em: <https://docs.microsoft.com/pt-br/cpp/standard-library/file-system-navigation?view=msvc-170>

Acesso em: 12 de fevereiro de 2022.

APÊNDICE A
Instruções de Compilação e Execução

Com a utilização de um arquivo Makefile, a compilação e linkagem do programa se dão de maneira muito simples:

Colocando o seu terminal no diretório raiz do projeto, basta que você digite `make` e o programa será compilado e linkado.

A execução se dá com seu terminal no diretório raiz do projeto e com o programa compilado, digitando no prompt:

```
bin/main -i <caminho/nome do documento com a consulta>
         -o <caminho/nome do documento com a saída produzida>
         -c <caminho/nome da pasta com os documentos do corpus>
         -s <caminho/nome do documento com as stop words>
```

Não é necessário que os parâmetros de entrada sejam passados nessa ordem, mas a lógica de flags deve ser respeitada.

Por fim, é possível excluir, caso se queira, os arquivos objects (`.o`) gerados com a compilação e, ainda, o arquivo executável, entrando com o comando `make clear` no terminal, quando ele estiver no diretório raiz do projeto.

APÊNDICE B
Arquivo de Stop Words do Teste 2

i
me
my
myself
we
our
ours
ourselves
you
your
yours
yourself
yourselves
he
him
his
himself
she
her
hers
herself
it
its
itself
they
them
their
theirs
themselves
what
which
who
whom
this
that
these
those
am
is
are
was
were
be
been
being
have
has
had
having
do
does
did
doing
a
an
the
and
but
if
or
because
as
until
while
of
at
by
for
with
about
against
between
into
through
during
before
after
above
below
to
from
up
down
in
out
on
off
over
under
again
further
then
once
here
there
when
where
why
how
all
any
both
each
few
more
most
other
some
such
no
nor
not
only
own
same
so
than
too
very
s
t
can
will
just
don
should
now