UNIVERSIDADE FEDERAL DE MINAS GERAIS
PROGRAMAÇÃO E DESENVOLVIMENTO DE SOFTWARE II - (PDS II)

Trabalho Prático: Máquina de Busca

Participantes:
Bernardo Teixeira de Amorim Neto
Emanuel Juliano Morais Silva
Marcos Vinicius Caldeira Pacheco

22 de novembro de 2019

1 RESUMO

Criamos, com base nos conhecimentos adquiridos no curso de PDS 2, uma Máquina de busca que consegue, com moderada eficiência, retornar a ordem de prioridade de documentos frente à query do usuário usando similaridade de cossenos e as técnicas adquiridas durante o curso, em particular os conceitos de orientação à objetos.

2 INTRODUÇÃO

Para criar a máquina de busca proposta no trabalho, dividiu-se o projeto em 4 grandes áreas, sendo elas:

- Leitura, criação e acesso do índice invertido
- Criação do sistema de coordenadas
- Comparação de ranking por meio de similaridade de cossenos
- Entrada da *query* do usuário e apresentação dos resultados esperados

Por meio dessa divisão foi mais fácil usar os conceitos de encapsulamento e orientação à objetos aprendidos no curso, facilitando, em particular, o trabalho em grupo, uma vez que foi possível manter os membros cientes apenas das funções, não de suas respectivas implementações. Além disso usou-se essa divisão para facilitar o processo de teste dos códigos, uma vez que cada teste foi escrito pelo próprio programador que implementou a função e priorizando, sempre que possível, isolar os testes de unidade, como foi proposto durante o curso.

Dessa divisão decidiu-se por criar 5 classes, sendo elas:

- IndiceInvertido: responsável por criar o índice invertido dos documentos da pasta “arquivos” e fornecer funções para acesso deste por classes posteriores
- Coordenadas: cria as coordenadas de um documento no eixo de uma palavra, utilizando, para isso, o “IndiceInvertido”.

- LeitorPesquisa: é responsável por receber a entrada do usuário e normalizá-la de acordo com as especificações do trabalho.
- Norma: calcula a norma de um vetor de coordenadas de um documento, serve de auxílio para a classe “Rank”
- Rank: recebe as palavras da pesquisa, já tratadas, e retorna um rank dos arquivos mais semelhantes.

Portanto foram usados diversas habilidades aprendidas no curso, incluindo o uso de “Makefile” e “git”, que facilitaram muito o trabalho em grupo, o encapsulamento e o processo de testes.

3 IMPLEMENTAÇÃO

3.1 ÍNDICE INVERTIDO

Sendo crucial na máquina de busca, a escolha da estrutura de dados para o para o índice invertido foi muito pensada, tanto do ponto de vista de eficiência quanto no que diz respeito à facilidade de implementação das outras classes, uma vez que um mal índice invertido dificultaria, e muito, todo o restante do trabalho. Dessa forma decidiu-se usar um *map*, que tem *strings* (palavras dos documentos) como chaves e *multisets*<*string*> (documentos que contém a palavra) como valores. Essa escolha se deu pois:

- Torna-se fácil e eficiente saber, à partir de uma palavra, quais são os arquivos em que esta está presente, dada a facilidade e eficiência do acesso no *map*. Esse acesso, inclusive, foi implementado na classe com a sobrecarga do operador “[]”, mantendo a sua função original, como é recomendado quando se faz sobrecarga.
- Também é fácil saber, a partir da estrutura de dados escolhida, os documentos em que a palavra aparece e o número de aparições. Essa foi a motivação da escolha do *multimap*, sabia-se que o acesso de documentos seria uma operação frequente nas classes posteriores, portanto optou-se por essa estrutura, uma vez que permite saber o número de vezes em que um elemento aparece em $\boldsymbol{O}(\log n)$.
- É fácil de implementar, uma vez que a inserção em um *map* é exatamente igual à sua entrada.

Para o restante da implementação, vale destacar algumas funções.

Funções:

As funções públicas usadas na implementação dessa classe são:

- **InserePasta(string pasta) -** essa função, que é a principal da classe, insere no índice invertido todos os arquivos presentes em “pasta”, usando para isso algumas funções privadas que, em suma: retornam os nomes dos documentos em “pasta”, lêem os documentos em “pasta” e formalizam as palavras retirando acentos e pontuações.
- **num_arquivos() -** retorna o número de arquivos totais que já foram lidos e inseridos no índice invertido.
- **nomes_arquivos() -** retorna o nome de todos os arquivos que já foram lidos e inseridos no índice invertido.

- **operator[string palavra] -** retorna um *multiset<string>* com os documentos que contém "palavra". Foi feita a sobrecarga pois tem a mesma função que a original na implementação do *map*.
- **todas_palavras() -** retorna um *set* com todas as palavras que já foram lidas e inseridas no índice, que são, portanto, todas as chaves do *map*.

Concluindo, a classe "IndiceInvertido" foi criada com o propósito de ser eficiente e simplificar seu uso pelas outras classes, uma vez que é o pivô dessa máquina de busca. Considera-se que isso foi bem feito, uma vez que foi a primeira classe a ser implementada e precisou de poucas alterações durante o andamento do processo, apenas se adaptando à necessidades que surgiram conforme o andamento do projeto.

3.2 COORDENADAS

A classe Coordenadas foi implementada de tal maneira que ela é capaz de retornar a frequência de uma palavra em um documento específico, bem como a importância dessa palavra no conjunto total de documentos, e o valor da um documento no eixo dessa palavra (por isso ela é uma coordenada). Dessa forma, optou-se por deixar cada instância "Coordenada" fixada com apenas uma palavra, ou seja, uma "Coordenada" representa todos os valores de qualquer documento, no eixo de uma única palavra.

Para realizar todos os cálculos necessários para encontrar esses valores, foi optado passar por referência o "IndiceInvertido" como parâmetro, juntamente com a palavra referente à "Coordenada". Desse jeito, todos os dados necessários para os cálculos conseguem ser facilmente obtidos.

O construtor da classe é responsável por definir todos as variáveis privadas que serão utilizadas. Uma variável que merece destaque é a *docsSemRepetir_*, um *double* que representa a quantidade de documentos diferentes em que a palavra está presente. Ela se destaca justamente porque para obtê-la foi criado um *set* a partir do *multiset* dos documentos em que a palavra aparece, e depois foi contado o tamanho desse novo *set*. Essa operação acabou sendo a mais custosa dentro da implementação dessa classe.

Funções:

As funções públicas usadas na implementação dessa classe são:

- **palavra()** - Retorna uma *string* da palavra (ou o eixo) a qual a Coordenada representa, utilizando uma variável privada.
- **frequencia(string doc)** - Retorna um *double* que é a frequência da palavra num documento específico, utilizando a função *count()* no *multiset* que tem todos os documentos em que a palavra aparece, com repetição.
- **importancia()** - Retorna um *double* que é a importância da palavra no conjunto total de documentos. Esse valor é obtido utilizando a expressão matemática mostrada nas instruções do trabalho (equação 4.1). Como essa equação utiliza logaritmo, foi incluída a biblioteca "math.h" nessa classe.
- **valor()** - Retorna um *double* que é a multiplicação entre a frequência de uma palavra em um documento pela sua importância. Essa função representa a equação 4.2 mostrada nas instruções do trabalho.

Concluindo, a classe Coordenadas foi feita para ser utilizada para se encontrar valores referentes a uma única palavra. Então, ao fazer uma pesquisa, essa classe deve ser instanciada para cada palavra (eixo) necessário para se realizar o ranking.

3.3 NORMA

A classe "Norma" foi utilizada como uma forma de otimizar o processo de pesquisa através de um pré-processamento.

Observando a fórmula da similaridade de cosseno:

$$sim(d_j,\ q) = cos(\theta) = \frac{\sum_{i=1}^{t}(W(d_j, P_i) \times W(q, P_i))}{\sqrt{\sum_i W(d_j, P_i)^2} \times \sqrt{\sum_i W(q, P_i)^2}} = \frac{W_{documento} \bullet W_{pesquisa}}{\|W_{documento}\| \, \|W_{pesquisa}\|}$$

É possível perceber que, para calcular a norma do vetor dos documentos, não necessariamente precisamos saber qual foi a pesquisa feita pelo usuário, assim, a função "Norma" pré-computa esses valores, utilizando o seguinte processo:

- Inicialmente, itera sobre todas as palavras dos documentos, para calcular suas coordenadas $(palavras)$;
- Depois, itera-se pelos documentos que contém cada palavra, calculando-se o valor da coordenada e somando à norma de seu respectivo documento $\boldsymbol{O}$
- Por fim, calcula-se a raiz do resultado achado para cada arquivo, salvando em um map $(documentos)$.

Sendo assim, no pior dos casos, esse algoritmo opera em $(documentos * palavras)$, complexidade similar às demais implementações.

Porém, tendo em vista o número de palavras existentes, pode-se assumir que os documentos serão esparsos, sem muitas palavras em comum, o que reduz drasticamente o tempo de execução.

Funções:

- **operator[string palavra] -** este *operator* foi implementado para facilitar o acesso às normas.

3.4 RANK

A classe "Rank" foi implementada para, dado um conjunto de palavras, retornar uma sequência de k documentos $(k = min(10, \#docs))$, tal que eles estejam ordenados de forma decrescente em relação à similaridade com a pergunta.

Tendo em vista a pré-computação feita pela classe "Norma", para calcular os valores de $W_{documento} \bullet W_{pesquisa}$ e $\|W_{pesquisa}\|$, tudo que precisamos são das entradas não nulas, ou seja, das palavras da pesquisa para todos os documentos.

Após esse processo, os resultados são ordenados.

Portanto, a complexidade desse algoritmo é de $(max(docs * log(docs), docs * pesquisa))$, um resultado bem mais eficiente, se formos considerar que as pesquisas são normalmente pequenas e o número de documentos também não deve ser muito grande.

Funções:

- **similaridade(string arquivo, multiset <string> &query, IndiceInvertido &indice, Norma &norma_arquivo) -** essa função calcula a similaridade de cosseno, a partir das fórmulas anteriormente vistas, iterando pelas palavras da pesquisa e por um documento em específico.
- **ord(pair<string, double> a, pair<string, double> b)** - função base utilizada no construtor do vetor "rank_". Seu papel é ordenar em ordem decrescente os documentos pela sua similaridade.
- **imprimir(int k) -** imprime na tela o rank dos documentos mais similares, permitindo ao usuário abrir algum de seu interesse.

Concluindo, a classe "Rank" une diversas classes para calcular as similaridades e mostrar os resultados ordenados da pesquisa ao usuário.

3.5 LEITOR PESQUISA

A classe "LeitorPesquisa" foi criada com o objetivo de receber as palavras da query, tratá-las (tirar suas pontuações e transformar letras maiúsculas em minúscula), remover palavras que não estejam presentes em nenhum documento e depois retornar um *multiset* de *string* com essas palavras.

Para isso foram usadas algumas funções muito simples, são elas:

Funções:

- **Pesquisa(IndiceInvertido &indice) -** é a única função pública da classe e basicamente chama as funções privadas da maneira correta. Recebe um "IndiceInvertido" como parâmetro para checar se a palavra pesquisada existe em algum documento, caso o contrário, essa palavra não é usada para construir coordenadas, uma vez que todas seriam zeradas. Após usar alguns procedimentos que serão explicados abaixo, esta função retorna um *multiset<string>* com as palavras da pesquisa.
- **entrada() -** recebe uma linha de entrada do usuário e retorna-a como string.
- **formaliza_palavra(string palavra) -** retorna uma versão normalizada de "palavra", sem acentos, letras maiúsculas ou pontuação.
- **frequencias() -** retorna o *multiset<string>* com as palavras da pesquisa já normalizadas e com sua existência verificada.

Concluindo, a classe "LeitorPesquisa" é uma classe muito simples e que serve apenas para formatar a entrada do usuário para ser usada na criação dos ranks, não sendo necessário preocupar-se com sua implementação, que também é muito simples.

3.6 MAIN

O "*main"* é responsável por unir todas as classes e permitir o funcionamento da máquina de busca, apesar de que nele não é feito nada além de chamar as funções e printar mensagens para o

usuário, uma vez que implementar qualquer coisa além disso na “main” seria imprudente, uma vez que esta função é particularmente difícil de testar. Na “main” são realizados os seguintes passos:

- Cria-se o “IndiceInvertido” utilizando todos os documentos dentro da pasta “arquivos”.
- Cria-se o “LeitorPesquisa” para receber a *query*.
- Cria-se um *multiset* com as palavras da *query* já tratadas.
- Cria-se o Rank a partir da pesquisa e do “IndiceInvertido” criado
- Imprime os 10 primeiros resultados
- Pergunta ao usuário se quer fazer mais querys e se quer abrir algum dos arquivos encontrados.

Dessa forma, fica claro que a “main”, além de comunicar com o usuário, apenas chama outras classes, diminuindo assim a chance de erros e aproveitando a ideia do encapsulamento.

4 TESTES

Para cada classe criada, foi adicionado um arquivo que utiliza a biblioteca *“doctest.h”* para testar cada uma de suas funções.

Cada criador da classe ficou responsável para criar os seu respectivo teste. Assim, um membro não precisaria saber da implementação de outro e poderia assumir a corretude do código dos outros. Os testes foram criados durante o processo e foram muito úteis durante seu desenvolvimento, de tal maneira que “make teste” tornou-se o padrão a cada compilação.

Sobre os testes em si, foi feito o máximo para isolar cada teste, inclusive alterando a implementação das funções para tal. Vale ressaltar que, em todos os testes das classes foi criada uma classe amiga para poder testar os métodos e variáveis privados.

5 CONCLUSÃO

Concluindo, foi criado uma Máquina de Busca que utiliza cinco classes diferentes, unidas no main, para receber o *query*, receber os documentos dentro da pasta “arquivos”, e imprimir na tela um ranking com a ordem dos documentos mais relevantes.

O trabalho foi organizado utilizando várias classes diferentes de forma a possibilitar que cada membro consiga implementar uma classe com uma certa independência uma da outra.

Também vale destacar que adquiriu-se, durante o percurso, bastante experiência em usar “git”, “makefile” e “doctest”. Todos os membros do grupo usaram o “GitKraken” para realizar os commits, o que facilitou a maneira com que mantinha-mos registro das modificações que cada um fez.

Esse trabalho prático não foi só um trabalho final, *foi mais do que isso*. Foi uma boa experiência para todos os membros do grupo, pois conseguimos compreender como funcionam e são organizados trabalhos de escala maior do que estávamos acostumados. Conseguimos também fixar melhor o conceito de orientação à objetos, uma vez que o encapsulamento, unido à confiança dada pelos testes, permitiram que outro programador usasse uma classe sem se importar com sua implementação, o que funcionou muito bem durante todo o processo.

“É mais do que isso”

~ Thiago, 2019