Documentação:

Trabalho Prático - Máquina de Busca

Programação e Desenvolvimento de Software II

Arthur Araújo Rabelo

Rafael Araújo Magesty

Lucas Araújo Magesty

Introdução

Em um primeiro momento, buscamos entender de uma maneira geral o que este trabalho demandava e quais seriam as estratégias e as dificuldades para implementá-lo, tanto no desenvolvimento do código quanto nas ferramentas externas: Git, documentação e divisão do trabalho.

Já no começo detectamos os pontos em que não estávamos familiarizados: a iteração sobre os arquivos de um determinado diretório, a criação de uma classe do zero (Índice Invertido), a Indexação e a normalização de palavras segundo uma regra específica. No entanto, tudo isso foi tranquilamente resolvido.

As maiores dificuldades surgiram em outras partes do código, principalmente nas funções do sistema de Recuperação, isto é, na parte que é responsável por retornar os documentos que contém todas as palavras consultadas, ordenando-os de acordo com o número de recorrências da consulta e, em caso de empate, lexicograficamente. A partir do sistema de indexação, tivemos muitos problemas em implementar as funções que organizam os elementos (documentos, *hits*, etc) para retornar o que o sistema visava com corretude.

No entanto, também foi possível transpor essas barreiras e, de maneira geral, esse projeto serviu como base de aprendizado não só para questões de lógica de programação, sintaxe e bibliotecas, mas também para outros fatores: a organização de um código, a sua modularização e a utilização das ferramentas do GitHub conectado ao VSCode. Percebemos que tais fatores são de extrema importância para a construção da nossa futura carreira como programadores.

Implementação

Os sistemas de uma Máquina de Busca se dividem em três. O que este projeto visa implementar são o subsistema de Indexação e o subsistema de Recuperação.

Subsistema de Indexação:

Para esta etapa, construímos a classe *indice_invertido.h*, que é a base dos dois subsistemas a serem implementados, visto que a recuperação só poderá ser feita quando a indexação já estiver devidamente organizada. As funções Inserir, Preencher, Pertence, Normalizar e Excluir foram declaradas para trabalhar em cima dessa base que armazena as palavras juntamente aos documentos aos quais elas pertencem e o número de suas recorrências em cada arquivo.

A função Normalizar transforma as palavras, tanto na consulta quanto na indexação, de acordo com o padrão definido no projeto. Ela recebe como parâmetro a string e a percorre por um iterador. A cada elemento da string, é checado se este pertence ou não ao padrão de caracteres válidos (letras minúsculas), armazenados no arquivo alfabeto.txt. Isso é feito por uma função externa find, que diz se um caractere pertence a determinado vetor. Se pertence às letras válidas, então o caractere é adicionado a string a qual função retorna. Se não, há duas opções: ou é uma letra maiúscula ou é um caractere que não deve ser inserido, tal como letras acentuadas ou caracteres especiais (@, !, &...). No primeiro caso, a letra é transformada em minúscula e inserida na string de retorno. No segundo, o caractere não é adicionado.

A função Inserir é responsável por adicionar um elemento na variável map<string, map<string, int>> indiceInvertido_. Ela recebe como parâmetro a palavra já normalizada e os documentos aos quais ela pertence, incrementando o número de recorrências toda vez que ela é encontrada num determinado arquivo.

A função Preencher tem por objetivo indexar todas as palavras dos documentos no índice invertido. Para isso, é necessário percorrer todo o diretório, arquivo por arquivo. Isso é feito principalmente com a biblioteca <*dirent.h*>. Essa função permite criar ponteiros que iteram sobre um diretório e métodos que retornam o nome dos arquivos, por exemplo. Com os nomes dos arquivos podemos usar a biblioteca <*fstream*>, mais comum na manipulação de arquivos. O código funciona assim: enquanto o ponteiro da <*dirent.h*> ainda não é nulo, ou seja, enquanto ainda existem arquivos na pasta, todos eles são percorridos, abertos por *ifstream* e lidos de tal forma que cada palavra lida é normalizada e inserida no índice invertido.

A função Pertence tem como objetivo verificar se a palavra digitada pelo usuário na Máquina de busca está contida nos documentos do diretório. Para isso utilizamos um *for* para percorrer todo o map<string, map<string, int>> indiceInvertido_, e comparar se a palavra é igual a string desse map. Adicionamos um contador para verificar se a palavra de fato foi encontrada: o contador incrementa sempre que a condição for verdadeira. Se o contador for maior que zero, então a palavra está no índice invertido e, portanto, a função retorna *true*. Do contrário, é retornado *false*.

A função Excluir limpa o índice invertido usando o método *.clear()* do *map*.

Subsistema de Recuperação:

A função Procurar é responsável inserir na variável *map<string, pair<int, int>> documentos_* os elementos necessários para o retorno final do código: o nome do documento (string) e o par que contém o número de hits da consulta e o número de palavras da consulta que foram achadas no documento. Para tanto, primeiro verificamos se a palavra recebida como parâmetro pertence ao índice invertido. Se pertence, iteramos sobre o índice invertido e, quando a palavra for encontrada, adicionamos o nome do documento, somamos número de recorrências dela no segundo elemento do *pair* e 1 ao segundo elemento do *pair*. Se as outras palavras pesquisadas pertencem ao mesmo documento, apenas os inteiros do pair são alterados, a fim de ter o número final de *hits* da consulta e ter o inteiro que serve para dizer se todas as palavras pesquisadas estavam no mesmo documento. Obs.: nesta função, foi utilizado o tipo *pair* porque ele permite alterar seus dois valores de uma determinada chave com facilidade. Se criássemos, ao invés do *pair*, um *map* dentro do *map documentos_*, isso não seria possível.

A função Imprimir por sua vez foi utilizada para retornar o nome do documento que se encaixa na busca feita pelo utilizador. Ela consiste na criação de um vetor com os pares de chave-valor do map *documentos_*, seguida da ordenação desse vetor usando uma função de comparação personalizada, que coloca os elementos com o maior número de hits primeiro. Esta função é a ComparaHits. A impressão do nome do documento na tela é feita através de um for que itera sobre o vetor e testa se o contador de palavras, recebido como parâmetro e que incrementa na consulta (conta o número de palavras consultadas), é igual ao segundo elemento do *pair* de *elementos_*. Se isso for verdadeiro, isso significa que todas as palavras consultadas estão no documento e, portanto, esse documento deve ser retornado. Então, é printado somente o nome do documento, como especificado no projeto. Obs.: A ordenação por ordem lexicográfica em caso de empate no número de hits já é automática no tipo *map*.

A função ExcluirDoc limpa a variável *documentos_*.

Main:

No main, criamos uma variável do tipo indice_invertido e a preenchemos com a função Preencher. Depois, é criado um loop que lê a linha da consulta digitada pelo usuário. Nesse loop, a variável que conta as palavras é inicializada. Dentro desse loop, é inicializado outro loop que lê palavra por palavra da linha consultada: o contador incrementa e a palavra

é lançada como parâmetro para a função Procurar. Depois que todas as palavras da consulta foram lidas, o resultado é retornado pela função Imprimir e a variável MaquinaDeBusca é limpada.

Exceções

As exceções lançadas foram: PalavraInexistente, que diz quando uma palavra não existe no documento, e ConsultaSemCorrespondencia, que diz quando não existe um documento que possui todas as palavras consultadas. A primeira é lançada quando o método Pertence é retornado como falso na função Procurar. A segunda é lançada na função Imprimir: quando a condição que testa se o número de palavras consultadas é igual ao número dessas palavras em um mesmo documento não é verdadeira, isso significa que não há nenhum documento que tem todas as palavras da consulta.

Instruções para compilação do código

Para compilar o código, basta digitar “make” para chamar o Makefile e, em seguida, “./tp_executavel”.

Conclusão

De maneira geral, acreditamos que a experiência com este trabalho foi bem produtiva, no sentido de que foi possível aprender bastante sobre programação em todos os seus âmbitos. Apesar de nos depararmos com certas dificuldades, a transposição delas por esforço e pesquisa nos deu confiança para seguirmos aprendendo mais sobre o desenvolvimento de sistemas e softwares.

Cremos que o resultado que esperávamos foi alcançado e que conseguimos fazer o que nos foi demandado nas especificações do projeto. Portanto, a experiência foi, no geral, satisfatória.