Fundação Getúlio Vargas, Rio de Janeiro
Trabalho Final — Estrutura de Dados e Algoritmos
Integrantes: Alexandre Garriga, Julia Queiroz e Victor Vilanova

Search Engine

18 de junho de 2020

1 Introdução

Nosso trabalho consiste na implementação do Search Engine (uma espécie de mini google), cujo objetivo é buscar uma palavra e retornar o documento em que esta palavra está contida.

2 Acesso ao projeto

Github: `https://github.com/victorvilanovab/project_SearchEngine`

Vídeo: `https://youtu.be/_1jzjf-gYFM`

Requisitos:

- Link para download dos documentos necessários para o projeto: `https://gvmail-my.sharepoint.com/:f:/g/personal/b39003_fgv_edu_br/EnaiGxh-msxDjfTcPZtGD1AB_CqPiyN78FfUPzdZrqyKGQ?e=eCDvks`
- No link acima, estão os arquivos de serialização da trie, a fim de que possamos fazer a deserialização com o nosso executavel (trie_execute.cpp) e os arquivos de documentos e títulos nas pastas devidas que são necessários para serem retornados ao usuário.

Instruções:
Para rodar o programa:

- Baixe todos os arquivos do link especificado acima clicando em "Baixar" no menu de tarefas do One Drive. Irá baixar um arquivo zipado que corresponde a pasta dos dados necessários para nosso projeto.
- Extraia a pasta do arquivo baixado.
- Dentro dessa pasta, deve conter uma pasta chamada (docs), uma pasta chamada (titles) e um arquivo chamado (serialization_docs_titles.txt).

- Dentro da pasta docs, deve conter 7 arquivos, 4 referentes a docs_p1 e 3 referentes a docs_p2. É necessário que estraia esses arquivos .rar nessa mesma pasta que vieram (colocar extrair aqui). Ficando ao final dentro da pasta docs 2 pastas. A primeira pasta chamada de docs_p1 que deverá conter 798.881 arquivos txt e a segunda pasta chamada de docs_p2 que deverá conter 560.331 arquivos txt.

- Dentro da pasta titles, deve conter 2 arquivos, 1 referente a titles_p1 e 1 referente a titles_p2. É necessário que estraia esses arquivos .rar nessa mesma pasta que vieram (colocar extrair aqui). Ficando ao final dentro da pasta titles 2 pastas. A primeira pasta chamada de titles_p1 que deverá conter 798.881 arquivos txt e a segunda pasta chamada de titles_p2 que deverá conter 560.331 arquivos txt.

- O código em C++ chamado trie_execute.cpp está no Github. Nele, está implementado a estrutura de dados e a função que faz a deserialização do arquivo de serialização. Assim, criando a nossa árvore.

- É necessário realizar alterações no código (trie_execute.cpp), indicando os paths corretos das pastas dos docs, titles e do arquivo de serialização. Assim, alterando os paths locais. Será necessário alterar as variáveis [path_docs], [path_titles] e [path_serialization_file]. O [path_docs] é o caminho onde estará as pastas (docs_p1 e docs_p2); O [path_titles] é o caminho onde estará as pastas (titles_p1 e titles_p2); E, finalmente, [path_serialization_file] é o caminho onde estará armazenado o arquivo (serialization_docs_titles.txt).

- Bibliotecas que são necessárias para o código (trie_execute.cpp): <iostream>, <vector>, <string>, <fstream>, <sstream>, <algorithm>, <typeinfo>, <chrono> e <iterator>

3 Descrição do projeto

Inicialmente, nós temos apenas os documentos da Wikipedia que estão distribuídos em 164 arquivos contendo 10.000 documentos cada. A partir desses dados, o programa foi desenvolvido em torno de 5 principais partes:

3.1 Limpeza dos dados

Primeiramente, optamos por separar os documentos em arquivos individuais, de modo que quando precisarmos abrir qualquer documento dentre os 1.640.000, podemos realizar isso em um tempo constante. Assim, separamos os documentos em título e corpo, pois visualizamos que precisaríamos retornar apenas os títulos.

Além disso, o corpo dos nossos documentos não estava preparado para o processamento, pois continha diversos caracteres que não pertenciam ao alfabeto

do ASCII. Dessa forma, realizamos uma limpeza em Python, por ser uma linguagem de fácil manejamento e por possuir ferramentas eficazes para esse tipo de trabalho.

Após a finalização da limpeza de dados, seguimos para a compreensão da estrutura de dados.

3.2 Estrutura de dados

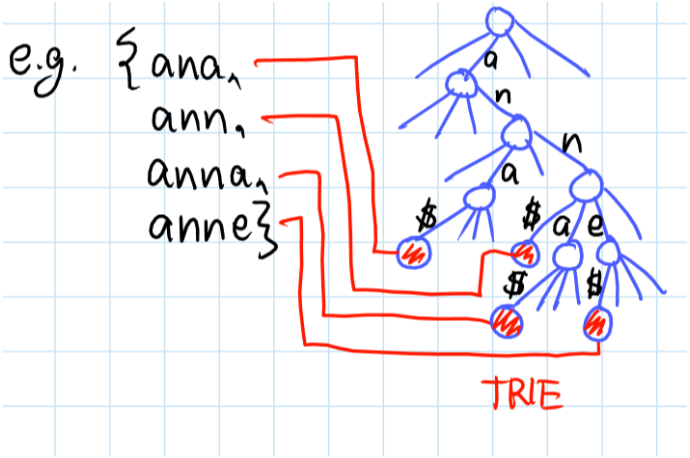
Para realizar a busca da query pedida, optamos por utilizar a estrutura de dados TRIE, que é uma árvore de prefixo das palavras. A implementação foi realizada em C++, pela sua eficiência e por ser requisito na composição do projeto.

Implementamos a TRIE utilizando ponteiros, de forma a minimizar o tempo de busca. Cada nó da TRIE é composto por 26 ponteiros, além de 1 booleano (usado para saber se é o fim de uma palavra) e 1 vetor (onde armazenaremos os documentos os quais cada palavra está contida).

Inicialização do nó: Todos os 26 ponteiros apontam para Null, boolean é FALSE e inicia um vetor dos documentos.

Assim, saem 26 ponteiros do nó raiz, que são as primeiras letras das palavras e de cada um desses ponteiros saem novos 26 ponteiros, que indicam o segundo caractere das palavras, e assim por diante.

Abaixo, podemos observar um exemplo da estrutura de dados escolhida, a fim de melhor compreensão do leitor.



Assim, da nossa foto, podemos ver que como todas as palavras do dicionário começam com **a**, do nó raiz sai apenas o **a**. Os outros 22 ponteiros não estão

aparecendo no desenho pois eles apontam para **Null**.

Também é importante ressaltar que a fim de ganhar eficiência no nosso código e na hora da busca, ao invés de indicar o char de cada letra em cada nó, optamos por indicar o index da letra do ASCII. Dessa forma sabemos qual ponteiro indica cada letra do alfabeto.

3.3 Implementação da TRIE

Após a realização da nossa estrutura, implementamos as duas funções principais: search e insert. Ambas percorrem nossa trie de acordo com um parâmetro string e realiza as devidas operações:

Ex: search("yes")
Raiz >> nó que corresponde à letra y >> nó que corresponde à letra e >> nó que corresponde à letra s.

Se o último for nulo, a palavra **yes** não existirá na TRIE. Caso contrário podemos realizar as seguintes ações:

Ex: insert("yes", número do documento)
Raiz >> nó que corresponde à letra y >> nó que corresponde à letra e >> nó que corresponde à letra s.

Caso o último seja Null, criamos o nó **s**, o boolean vira True e é adicionado o número do documento ao final do vetor documentos desse nó.
Sendo assim, nossa trie estava tomando forma, e então começamos a realizar treinamentos com parte dos dados e criar um menu no console de forma a interagir com a nossa Search Engine.

3.4 Serialização

Após o processo da implementação da TRIE, obtemos um tempo de aproximadamente 8 horas para construir a TRIE e assim realizar as buscas com todos os 1 milhão de documentos.

Nesse sentido, foi preciso realizar um pré-processamento nos dados, ou melhor, uma serialização da TRIE a fim de otimizarmos esse tempo de inicialização. A serialização foi feita de forma a armazenar em um arquivo texto, cada palavra com seus respectivos documentos. Desse jeito, poderemos antecipar todo o trabalho de abrir cada documento e portanto, inicializar a TRIE de forma sucinta.

3.5 Interface do console

Foi preciso fazer uma interface de forma intuitiva para manejamento da realização das querys. Assim, implementamos um menu principal onde podemos

escolher se queremos realizar uma query ou sair do programa.

Ao escolhermos realizar uma query, o programa pede para digitar a(s) palavra(s) desejada(s) e retorna os títulos de 20 em 20 e pergunta se deseja abrir mais títulos. Caso não, o programa pergunta se deseja abrir um dos títulos. Caso não novamente, retorna ao menu principal.

4 Extras

4.1 Auto suggest

Implementamos um autosuggest para facilitar a query quando a mesma é digitada de forma incompleta.

Por exemplo, ao pesquisar "admi", retornamos palavras como administração, administrativo, admirável e afins, além de uma interface que permite que o usuário escolha uma dessas palavras para query.

4.2 Ranqueamento por títulos

No nosso programa, um documento que tinha a palavra pesquisada apenas uma vez tinha a mesma importância que aquele que a palavra pesquisada era o tema principal do documento. Por esse motivo, tivemos a ideia de realizar um ranqueamento por títulos.

Para isso, foi preciso realizar a TRIE também com os arquivos de títulos, a fim de armazenar nas palavras os documentos cujas palavras destes títulos pertencem.
Desse modo, os documentos retornados para o usuário primeiro retornam aqueles cuja query pesquisada pertence ao título, após isso, retornam os outros documentos que a palavra pesquisada pertence apenas ao corpo do documento.

Concluímos que essa ideia facilitou nossa busca por palavras de modo que tornou a pesquisa na nossa TRIE mais eficaz para o usuário, pois os primeiros documentos retornados estão diretamente relacionados com a pesquisa.

5 Resultados

Tempo de inicialização: 6 minutos

Tempo de query: 0,0016 segundos

Tempo de print dos documentos: instantâneo (constante)

Ao nosso ver, os tempos de execução foram bastante eficazes. O tempo de inicialização é de aproximadamente 6 minutos dependendo das especificações do computador. Ademais, o tempo de busca é extremamente rápido, em instantes ele varre a trie para saber se existe a palavra buscada, e o tempo aumenta um pouco dependendo do tamanho da palavra ou frase.

Para rodar nossa trie, o programa lê o arquivo txt e cria a árvore. Nessa etapa, o que nosso programa está fazendo é retirando a informação armazenada no HD e transferindo-a para a memória RAM.

Ao final, é requisitado pelo nosso programa aproximadamente 3,5GB de memória RAM. Acreditamos que é um resultado satisfatório e que qualquer usuário não terá dificuldades de rodar nosso programa.

6 Limitações e trabalhos futuros

De limitações, podemos citar que nosso código é limitado a rodar em um computador, logo não sendo possível de rodar em um smartphone, por exemplo.

Nesse sentido, como trabalho futuro, uma possibilidade seria a criação de um aplicativo de celular de buscas na Wikipedia. Uma outra possibilidade é a criação de uma página web para qualquer usuário com acesso à internet utilizar nossa aplicação.

7 Conclusão

Embora a apresentação final do trabalho tenha ficado de forma simples, acreditamos que esse trabalho foi de grande valia e aprendizado para realização de projetos de grande porte.

Aprendemos a dividir tarefas e organizar nossos afazeres de maneira a maximizar os resultados obtidos. Dessa forma, foi bastante enriquecedor elaborar esse trabalho.

8 Distribuição do trabalho

Limpeza dos dados: Julia Queiroz

Implementação da TRIE: Alexandre Garriga e Victor Vilanova

Interface no console: Alexandre Garriga e Victor Vilanova

Extras: Julia Queiroz e Victor Vilanova

Relatório: Julia Queiroz e Victor Vilanova

Vídeo: Alexandre Garriga, Julia Queiroz e Victor Vilanova

Apresentação: Julia Queiroz