Googlezito

Carolina Monteiro, José Arthur Souza e Laura Chaves

June 2020

Abstract

Nosso grupo montou uma ferramenta de busca no corpus da Wikipédia em inglês. Para isso, montamos uma trie, na linguagem C++, fizemos a limpeza dos dados do corpus em Python, e a busca em si em C++ também. Também foi implementada a serialização e desserialização da árvore, assim como um mecanismo de sugestão de palavras caso a busca não encontre sucesso. Chamamos nossa ferramenta de Googlezito.

1 Agradecimentos

Gostaríamos de agradecer ao nosso monitor Giovani Valdrighi, que ajudou muito todos do grupo quando precisávamos de orientação.

2 Referências

O GitHub do repositório onde o grupo pode desenvolver a ferramenta de busca pode ser acessado pelo link `https://github.com/josearthursouza/googlezito`

Os pacotes do C++ que são utilizados são:

- locale.h
- chrono
- algorithm
- iostream
- fstream
- sstream
- string
- vector

Além disso, deve ser incluído no início do código a seguinte linha:

using namespace std;

Para poder usar o mecanismo de busca Googlezito, é necessário:

- Fazer o download do arquivo GrandiosaArvore pelo link `https://drive.google.com/drive/folders/1E9Z3Ga5K_dihCYoZBdBSnsQbq7zcpqLP?usp=sharing`
- Faça o clone do repositório `https://github.com/josearthursouza/googlezito`
- Escolher uma das opções abaixo e executá-la:

  Opção 1:

  - Abra a pasta Raw do Wikicorpus, caso tenha ela baixada em seu computador
  - Apague os arquivos vazios (de tamanho 0KB)
  - Selecione todos os arquivos ao mesmo tempo e renomeie-os digitando "amigavel". O resultado vai ser 154 arquivos intitulados de "amigavel (1)" até "amigavel (154)"
  - Cole os arquivos "primeiraparte.py" e "segundaparte.py" do repositório na pasta onde estão seus arquvios "amigavel (i)"
  - Execute o "primeiraparte.py". Vão surgir 137 arquivos "conteudos˙desordem (1)" a "conteudos˙desordem(137)" e um arquivo "conteudos˙desordem". Pode apagar os arquivos amigavel se quiser.
  - Execute o "segundaparte.py". Vão surgir 137 arquivos "conteudos˙ordem (1)" a "conteudos˙ordem (137)" e um arquivo "titulos˙ordem". Pode excluir os arquvios de conteudos˙desordem e o titulos˙desordem se quiser.

  Opção 2:

  - Faça o download dos arquivos "conteudos˙ordem (1)" a "conteudos˙ordem (137)" e o arquivo "titulos˙ordem" pelo link `https://drive.google.com/drive/folders/1GItSYjyurEp8l9lLTdMHGSHajmb19GY5?usp=sharing`

- Junte numa só pasta todos os arquivos de "conteudos˙ordem (i)", o arquivo "titulos˙ordem", o arquivo "GrandiosaArvore" que baixou do Drive, e os arquivos "teste.cpp" e "teste.exe", clonados do repositório.
- Execute o arquivo "teste.exe" como preferir
- Faça sua busca, mas atenção! Nosso mecanismo não aceita letras maiúsculas nem símbolos especiais como acentos, os qualquer coisa que fuja do alfabeto de a-z e os números de 0-9.

O link para o vídeo com a demonstração do projeto é `https://youtu.be/lXMDEps-ICk`

3 Descrição

Começamos o trabalho baixando o corpus da Wikipédia, versão raw, pelo link fornecido pelo professor (`http://www.cs.upc.edu/~nlp/wikicorpus/`). A partir daí, começamos a pensar em que tipo de estrutura usaríamos, e de que forma começaríamos o pré processamento dos arquivos, dado que seria completamente inviável não fazer uma limpeza de qualquer tipo.

Nesse pré processamento, começamos apagando todos os documentos vazios que vieram da pasta raw. Depois, todos os artigos foram renomeados de forma que fossem mais facilmente acessados e compreendidos pelo grupo. Daí, utilizando a linguagem Python, todos os títulos foram separados na ordem em que aparecem nos documentos originais e foram escritos em um novo documento de texto, no qual cada linha era um título, na ordem que aparecia. De maneira análoga, cada texto foi separado, também na ordem em que aparecem originalmente, em documentos à parte, com um máximo de 10 mil textos por documento. Dessa forma, ficamos com 137 arquivos com 10 mil textos cada, e um arquivo com todos os títulos, na mesma ordem no qual os textos estão organizados nos 137 arquivos.

A partir daí, todos os títulos foram ordenados em ordem alfabética, usando o algoritmo de mergesort, e depois os textos foram ordenados de forma a continuarem associados aos seus títulos correspondentes. Resultou em um novo arquivo com os títulos em ordem alfabética, e novos 137 arquivos com 10 mil textos cada, mas respeitando a ordem alfabética de seus títulos dessa vez.

Depois, começamos o processo de preparar os arquivos textos para serem lidos para inserção na árvore. Cada texto foi limpo, eliminando palavras com caracteres fora do alfabeto que estamos usando (26 letras + 10 algarismos), assim como sinais de pontuação. Criamos um dicionário (estrutura do Python), no qual cada chave era uma palavra presente no corpus e o valor associado era uma lista com os índices dos artigos no qual a palavra aparece. Daí, as palavras do dicionário e seus valores foram escritos em novos arquivos de textos para serem lidos na estrutura.

Inicialmente, pensamos em trabalhar com a estrutura map do C++, que funcionaria como uma espécie de dicionário, mas o grupo acabou por decidir fazer a trie, pois cremos ser o melhor método, levando em conta o menor tempo de criação e implementação que esta estrutura propõe.

Nessa árvore, inserimos palavras da seguinte forma: começamos num nó raiz, que não possui valor, mas que está associado a um vetor de 36 elementos. Cada um desses elementos é um ponteiro para um possível filho (as 26 letras do alfabeto latino e os 10 algarismos indo-arábicos que utilizamos). A inserção de palavras se dá caracter por caracter. A cada caracter que inserimos, cria-se um novo nó, filho do caracter anterior na palavra, e isso é definido ao colocar um ponteiro a esse novo nó na posição correspondente do vetor de ponteiros do nó pai. Assim, cada letra de cada palavra é um nó, com um vetor de ponteiros associados aos seus possíveis filhos. Além disso, cada nó possui um indicativo de se há a palavra indicada nos arquivos do Corpus, ou seja, se a palavra existe nos textos da Wikipédia. No caso positivo, também é associado ao nó um vetor

com os índices de cada artigo do corpus no qual a palavra aparece. Assim, a busca ocorre por, começando no nó raiz, analisando cada caracter e descendo os níveis até atingir o fim da palavra, e daí verificando se a palavra existe na árovre e se existe no corpus da Wikipédia.

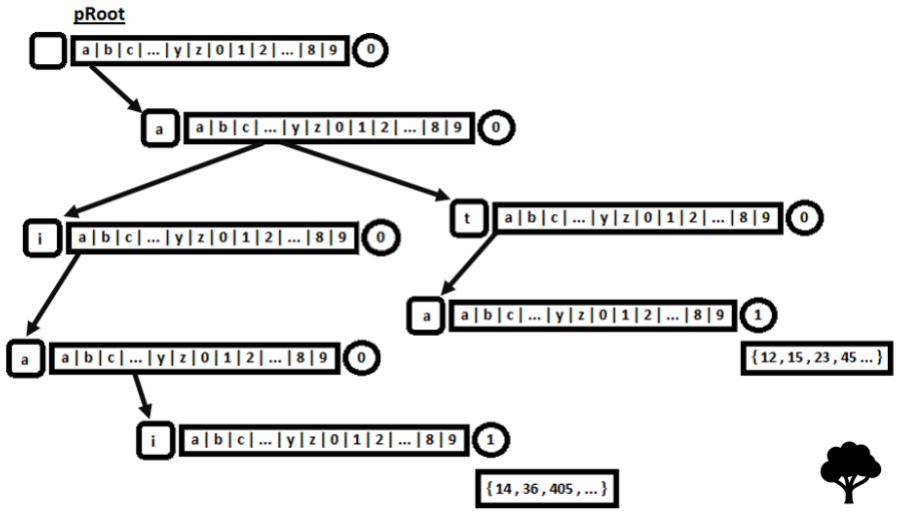

Figure 1: Ilustração de uma árvore com as palavras "aiai" e "ata" inseridas e seus títulos associados

Dada a árvore construída, também implementamos uma serialização em texto, para que não seja necessária fazer a construção da trie a cada vez que o programa rode, pois seria um processo muito lento. Essa serialização se deu por, considerando a estrutura montada, passar por cada nó e traduzindo para o documento de texto as informações que ele possui. No nosso caso, primeiro escreve o dado do nó, o caracter a que ele corresponde. Ao lado, escreve-se o seu booleano "fim", que é um indicativo de se é um fim de palavra ou não. Caso seja, o tamanho do vetor de índices de textos associados, ou seja, a quantidade de artigos no qual aparece, também é posta no documento, seguida dos índices em si. Para indicar que findou-se a listagem de índice, escrevemos um "-", que será importante na desserialização.

A desserialização ocorre num processo quase perfeitamente inverso à serialização. Lê-se o documento de texto, e suas informações são traduzidas à arvore sabendo que começamos sempre com o dado do nó, criamos um nó com o valor; depois será indicado se é fim de palavra ou não, associamos o booleano ao nó; caso seja fim de palavra, criamos o vetor de índices associado; e por fim, vemos cada filho que ele possa ter.

Ao executar o código, a primeira ação é desserializar, logo depois, o usuário pode fazer sua pesquisa. Cada palavra que o usuário digitar é diretamente buscada na árvore, pelo processo descrito nos parágrafos anteriores. Caso seja encontrada correspondência, mostramos ao usuário o tempo que levou para

fazer a busca em segundos e microssegundos, em quantos títulos a palavra foi encontrada e os resultados em que foi encontrada, em ordem alfabética. Caso haja mais de 20 títulos associados, a exibição dos resultados será feita de 20 em 20 títulos. Na situação de a palavra buscada não existir, o programa oferece algumas sugestões. O tempo de pesquisa é linear, variando com o tamanho da palavra pesquisada, e não é afetado pela quantidade de palavras inseridas na trie. Além disso, há no final um tratamento para que os textos da Wikipédia possam ser lidos em unicode, e não apenas com o alfabeto ASCII, padrão do C++.

A sugestão de palavras se dá por, ao identificar que a palavra pesquisada não existe, percorrer o maior caminho possível que inclua a palavra buscada. Pode ocorrer de a palavra existir por inteiro sendo parte de uma palavra, como no caso de pesquisar a palavra "aia" na árvore que coloquei de exemplo. Nessa situação, percorreria a palavra inteira e retornaria "aiai", pois veria que tem como continuar a palavra na árvore. Mas caso a palavra por inteiro não exista na árvore, ainda pode acontecer de parte dela ainda estar. Novamente, percorre-se o maior caminho possível usando a palavra pesquisada e, quando não for mais possível encontrar uma correspondência, retorna a palavra que possui maior parte da palavra pesquisada.

Caso seja feita a busca de mais de uma palavra, há um processo de comparação de títulos, ou seja, analisar o vetor de índices de títulos de cada palavra e comparar os dois, procurando quais os índices comuns aos dois. O processo no qual isso acontece é elementar, percorre-se os dois vetores e comparam os elementos de maneira inplace. Sabendo que, necessariamente, os vetores estão ordenados, a comparação começa comparando o primeiro elemento de ambos os vetores. Caso sejam iguais, apaga-se do segundo vetor (vec2) e compara o próximo elemento do primeiro vetor (vec1) com o novo primeiro elemento de vec2. Caso não sejam iguais, analisa qual dos dois elementos é menor, apagamos da lista do seu vetor correspondente e fazemos a comparação com o próximo. Isso ocorre até o primeiro elemento de vec2 ser maior que o último elemnto de vec1, ou até um dos vetores ficar vazio.

4 Resultados

No total, havia aproximadamente 1 milhão e 360 mil títulos de artigos, que separamos em 137 documentos de texto com 10 mil artigos cada. Nosso arquivo txt com a árvore serializada ocupa aproximadamente 1,81 gigabytes, e a pasta que contém os documentos necessários para rodar a ferramenta a partir da desserialização ocupa 5,72 gigabytes.

O processo de montar a árvore, com todos os 1 milhão e 360 mil artigos do corpus da Wikipédia, para que ela seja serializada dura aproximadamente 4 minutos, assim como a serialização, com o processo de desserialização sendo um pouco mais rápido, levando mais ou menos 200 segundos. Mesmo com pouca diferença no tempo, ainda é vantajoso fazer a desserialização, pois os documentos necessários para rodá-la ocupam menos espaço do que os documentos para

montagem da árvore.

O tempo de pesquisa não é constante, varia linearmente com o tamanho da palavra, mas as buscas mais demoradas não passam da ordem de 0,001 segundos. Buscas por mais de uma palavra também contam o tempo de comparação de títulos nos quais ambas as palavras aparecem, que variam com a quantidade de títulos que cada palavra aparece. No caso de duas palavras, uma que aparece em m artigos e outra que aparece em n artigos, a comparação de títulos tem tempo de execução da ordem de $O(n + m)$.

Palavra	Quantos artigos aparece
israelism	36
peruvian	1953
peru	6152
brazil	15339
care	27539

Figure 2: Tabela de algumas palavras e a quantidade de títulos no qual aparecem

Busca	Tempo de comparação (segundos)
israelism peru	0,003246
peru peruvian	0,004892
brazil israelism	0,020843
brazil peru	0,051045
israelism care	0,062517
brazil care	0,085939

Figure 3: Tabela de algumas pesquisas feitas com as palavras da tabela anterior e o tempo de comparação de títulos

5 Limitações

Por complicações envolvendo tempo e as condições mundiais nas quais nos encontramos, não conseguimos implementar tudo que tínhamos imaginado no começo do semestre.

Não conseguimos fazer uma interface web para nossa busca, um dos pontos principais que deixamos de incluir. Sentimos falta pois é uma forma de aproximar o Googlezito ao usuário, já que seria como acessar uma página na internet como inúmeras outras, ação ao qual o consumidor já está acostumado. Sabemos que executar programas direto pelo computador é menos palatável, porém

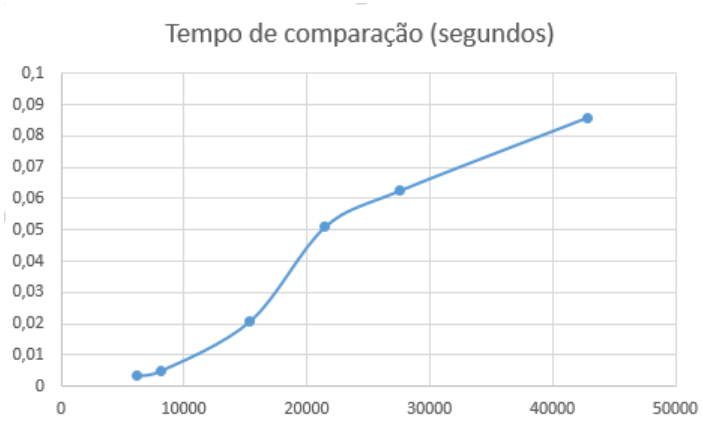

Figure 4: Gráfico que plota o tempo de comparação levando em conta a soma dos elementos do vetor de índices de cada palavra buscada. Notamos que é quase linear, com uma pequena anomalia em torno dos 20000 índices. Cremos que ela se dá por causa da grande discrepância nos valores de m e n.

compensamos a falta da interface com uma pesquisa extremamente rápida e eficiente.

Outra limitação do nosso trabalho é a busca de mais de uma palavra ao mesmo tempo. O algoritmo implementado, apesar de correto, pode ser bastante lento no caso de busca por duas palavras com muitas ocorrências cada.

Nossa sugestão também é um pouco falha, pois como é necessário percorrer a árvore para encontrar sugestões, caso o erro seja muito no começo, a sugestão não será muito precisa.

Além disso, não há um pré tratamento da busca feita pelo usuário, significando que buscas com letras maiúsculas ou qualquer caracter que não seja uma das 26 letras do alfabeto ou um dos 10 algarismos indo-arábicos não são computadas corretamente, e não darão o resultado esperado.

6 Trabalhos futuros

Em trabalhos futuros, pensaríamos principalmente em como contornar algumas das limitações que tivemos. Uma das principais seria fazer o pré processamento da busca feita pelo usuário, pois evitaria uma grande quantidade de erros na busca por motivos tão básicos, como buscar por palavras com letras maiúsculas ou acento. Também gostaríamos de implementar uma sugestão que fosse mais precisa, usando possivelmente um algoritmo de distância entre palavras para encontrar correspondências mais próximas. O tempo de sugestão de palavras, embora bom, não é ótimo, uma possivel otimização dele é algo que buscaríamos também. Além disso, uma possibilidade para melhorar o tempo da comparação de títulos seria começar a comparação pelos vetores com menos

índices.

Cremos também que a implementação de uma interface web seria também de extrema importância, pois torna a relação da ferramenta com o usuário bem mais amigável, tornando-se assim mais atrativa. Por fim, nos foi sugerido que, ao printar o artigo que fosse resultado de uma busca, utilizássemos quebra de linha, e não colocar o texto todo numa linha só. É uma característica facilmente implementada e que não está simplesmente porque não achávamos que seria necessário, mas caso fizéssemos novamente, levaríamos essa sugestão em consideração e implementaríamos.

7 Conclusão

Dessa forma, o grupo conseguiu desenvolver uma estrutura de dados inteiramente em C++ que faz pesquisa no corpus da Wikipédia em inglês em menos de um segundo, com o usuário podendo abrir qualquer um dos títulos encontrados caso queira e também recebendo sugestões de pesquisa caso a sua busca seja infrutífera. As mensagens retornadas pela demo são em espanhol, como forma de homenagem ao nosso professor Jorge Poco e o grande carinho que ele possui pelo seu país, Peru.

Para conseguir fazer isso, um pré processamento dos textos do corpus foi feito na linguagem de programação Python, que tratou todos os caracteres fora do alfabeto que utilizamos e também guardou as informações de maneira que fosse útil. Além disso, implementamos, em C++, uma trie, que permite ao usuário fazer a busca das palavras e também acessar os artigos aos quais a palavra buscada pertence.

O resultado foi uma busca que trabalha de forma extremamente rápida, com a montagem da árvore e a desserialização rápidas também, mas que não é perfeita. Encontramos algumas limitações no caminho, porém mesmo assim acreditamos que o trabalho foi um sucesso, pois cumpre a meta fornecida de forma eficiente e também proporcionou a todo o grupo um grande aprendizado de como criar e trabalhar com grandes estruturas de dados.

8 Distribuição do trabalho

- Carolina Monteiro: Foi a responsável por todo o pré processamento dos dados do corpus em Python, assim como pela inserção de tais dados na trie e de correção de bugs da implementação em C++, auxiliando José Arthur.
- José Arthur Souza: Foi o responsável pela implementação da estrutura da trie desde o começo, assim como de todas as funções da classe da busca.
- Laura Chaves: Responsável pela serialização e desserialização, utilização do unicode para leitura dos arquivos, elaboração deste relatório e também a única que conseguiu rodar arquivos tão pesados no computador pessoal.