Search Engine

Pedro Castilho Dall, Eduardo Vianna e Yolanda Nogueira

June 2020

1 Objetivo

O objetivo do projeto era a criação de um search engine para pesquisarmos na wikicorpus (língua inglesa) dando como entrada para o programa a(s) palavra(s) que queremos e recebendo como resposta os resultados de possíveis leituras que contém tal(ais) palavra(s).

2 Estrutura de dados

Utilizamos a Trie como estrutura de dados, que é construida a partir da criação de palavras. Nossa Trie é composta por:

- número de documentos;
- array de ponteiros que apontam para para os filhos;
- ponteiros para um array de vetores.

Esse último ítem ajuda a enriquecer a estrutura.

3 Detalhes da Trie

Tomemos a palavra "emap" como exemplo e o último nó (letra "p") com endereço X123.

Cada nó da árvore carrega consigo o endereço para o array de vetores, o número de elementos (conseguimos achar o documento apenas somando no endereço - X123 no exemplo) e um elemento que indica a próxima letra (no caso do último nó, vazio - exemplo do nó de "p").

3.1 Array de vetores

O primeiro elemento do array é a identificação do documento (k) que contem a palavra procurada ("emap"). Os outros elementos são as posições da palavra no documento k.

4 Pesquisa na Trie

Exemplo de busca: "matemática aplicada".

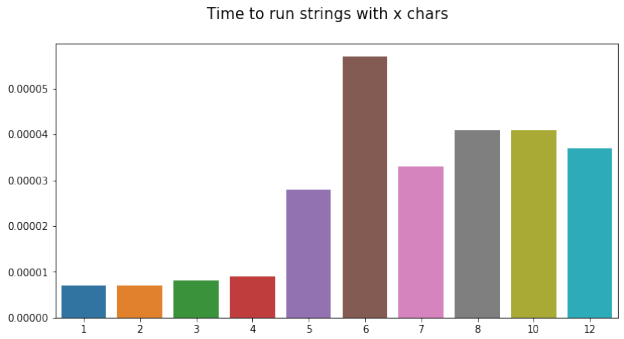
Nesse caso de mais de uma palavra, separamos em "matemática" e "aplicada" como primeiro passo. Em seguida, buscamos por cada palavra separadamente na árvore. Fazemos essa busca através da transformação de cada letra em seu número da tabela ASCII. Após acharmos o último nó da 1^a palavra (a palavra inteira), criamos um vetor de vetores que contém o endereço de cada vetor documento/posição correspondente a

palavra da busca, como a busca na Trie precismos percorrer carácter por carácter tempos que o tempo para encontrar uma palavra é linear com o número de caracteres. Buscamos a segunda palavra da mesma forma e armazenamos o vetor com os elementos correspondentes na linha abaixo da palavra procurada anteriormente (para fins de comparação).

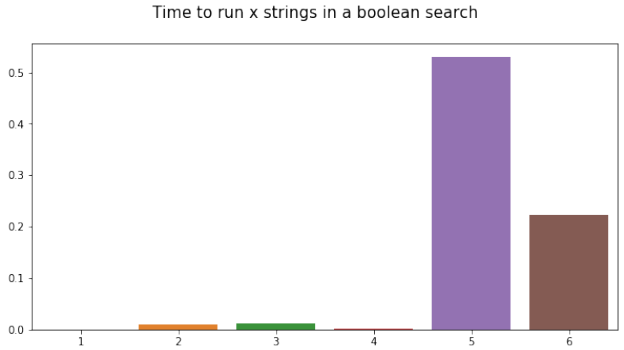
Em seguida, deletamos os elementos sem "par" da primeira linha e encaixamos (reorganizamos) os da segunda linha. Assim, fazemos a intersecção dos documentos que aparecem "matemática" e "aplicada". Para fazermos isso, criamos um for loop do ultimo elemento da primeira consulta ate o primeiro e um while (reverso também) para os elementos da segunda palavra, enquanto os elementos da primeira palavra forem maiores que o elementos do segundo vetor vamos deletando eles. Continuamos assim ate percorrer por todo o vetor e rodamos um ultimo for deletando os que sobraram. Esse algoritmo funciona pois nossos vetores ja estavam ordenados e ele roda em tempo linear. Como os vetores continham ponteiros para vetores o que estávamos comparando de fato eram os primeiros elementos de cada um desses vetores, terminado esses elementos estarão alinhados na matriz e, finalmente, cada coluna corresponde ao mesmo documento.

Entretanto, apenas o primeiro elemento de cada vetor representado na coluna é igual para as duas palavras (número de documentos). Os outros são diferentes, pois são as posições de cada palavra no texto. A vantagem dessa característica é que agora temos a possibilidade de fazermos buscas frasais e rankeamento baseado na frequência das palavras.

Abaixo temos alguns gráficos de pesquisas tanto analisando tempo com a quantidade de caracteres quando com múltiplas palavras:



Temos um comportamento um pouco estranho no gráfico e acreditamos que o código cria um cache para otimizar a pesquisa. Isso explicaria o porque de uma palavra com 12 caracteres demorar menos que uma com 6.



A pesquisa boolena é um pouco difícil de analisar por gráficos já que temos essa influencia do cache e como estávamos usando palavras aleatórias do vocabulário do corpus algumas intersecções podiam ser mais demorada que outras dependendo do numero de documentos que continham elas.

5 Limpeza

Para entender a limpeza dos documentos, pense que temos uma "caixa" cheia de arquivos com vários textos. Para nos organizarmos, colocamos todos os textos de cada arquivo em uma lista de listas. A primeira lista tem todas as linhas correspondentes ao primeiro testo, a segunda lista ao segundo e assim por diante.

Depois disso, criamos funções que procuram o titulo e a id de cada um desses textos. Assim, podemos criar uma lista com uma tupla de título + texto (ordenaremos todas essas linhas).

Depois de ordenados, colocamos todos os títulos em um arquivo .txt cujas linhas são os títulos.

As id's passam por uma limpeza com expressões regulares (útil para mexer em textos - adicionar, filtrar, etc). Utilizamos a biblioteca unidecode para tirar acentos e símbolos estranhos.

Após a limpeza, ainda temos a lista de tuplas. Carregamos para um .txt cujas linhas serão textos inteiros e estarão prontas para a serialização.

Pegamos todos os textos baseados em suas id's e separamos esses textos em novos "pacotes" de tamanhos 10000 organizados por ordem. Ou seja, os primeiros 10000 textos estarão em um .txt, os próximos 10000 em outro .txt ... A utilidade é que não precisamos passar por todas as linhas procurando o texto desejado: vamos direto para ele.

6 Serialização e Desserialização

A parte de serialização se da no arquivo trie.cpp onde recolhemos os dados de um arquivo .txt que contem em cada uma de suas linhas um texto do wiki corpus, trie.cpp entao devolve um arquivo serializado pronto para ser usado pelo nosso search engine.
A serialização é feita da seguinte forma começamos na raíz da Trie e, quando descemos um nó, sabemos tudo o que estamos acessando:

- em qual nó estamos (letra);
- quantos documentos tem dentro desse nó;
- id do documento;
- posições do elemento dentro desse documento;

Para o programa saber quando acabam as posições de um documento e começam as de outro utilizamos um sinal ("—"). Esse detalhe é fundamental para a desserialização. Assim, conseguimos diferenciar o nmero do documento das posições.

Para andarmos pela Trie descemos pelos nós à esquerda. Se não tiver, conferimos se existe algum nó à direita. Caso não haja nó nem à esquerda e nem à direita, subimos para o último nó visitado. Dessa forma, percorremos toda a árvore.

Na Desserialização construímos nó por nó assim como serializamos, no mesmo estilo de busca em profundidade. Como armazenamos todo o índice posicional nosso arquivo .txt ficou muito grande (4.2 Gb) e acabou aumentando bastante o tempo para preparar a arvore em torno de 650 segundos.

7 Limitações

Nossa busca só é realizada com termos em minusculo e sem acento, algo não muito complicado de se adicionar no futuro.

Por mais que nossa estrutura fosse rica o suficiente para buscas frasais e ranqueamentos baseado na frequência da palavra no texto tivemos problemas no final ao tentar implementar a busca frasal, algum erro de segmentação dado em uma linha "cout<<endl;" nos atrasou em uma semana e acabou faltando no nosso trabalho.

Justamente por ter mais estrutura nosso arquivo serializado ficou muito pesado e demorado pra deserializar, algo que poderiamos melhorar em futuras implementações.

8 Implementação Web

Conseguimos implementar uma interface web com algumas restrições, rodas uma parcela bem menor dos arquivos textos.

Aproveitamos a implementação web para ao escolhermos um titulo nos direcionar para a pagina da Wikipédia correspondente ao nosso titulo

9 Conclusão

Tendo o código e todos os arquivos, mandamos uma pesquisa como entrada e obtemos como saída as posições da busca. Em seguida, ele encontra as posições no título. Depois, retorna os títulos que que contem os textos com a nossa busca. Assim que escolhemos o título, o programa procura na pasta de textos e nos devolve o texto que queremos.

10 Distribuição do trabalho

- Serialização e deserialização: Pedro Dall e Yolanda Nogueira
- Busca na árvore: Pedro Dall
- Limpeza: Pedro Dall e Eduardo Vianna
- Iterface web: Eduardo Vianna e Yolanda Nogueira
- Vídeo: Pedro Dall, Eduardo Vianna e Yolanda Nogueira (edição e desenhos)
- Documento escrito: Yolanda Nogueira
- README (github): Pedro Dall e Eduardo Vianna