Trabalho Prático 1 de Algoritmos II

[**Mateus Jesué, Rafael Sant'Ana**]

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

rafael-sant-ana@ufmg.br, 2023087834
mateusjesue@ufmg.br, 2023099182

1 Introdução

Essa documentação foi feita para lidar com o problema da implementação de um motor de busca implementado em `Python`. O motor de busca foi implementado com o objetivo de avaliar consultas booleanas feitas em uma base de arquivos pré-determinada e retornar para o usuário, através de uma interface interativa, o resultado de sua consulta na base de arquivos existentes. Para atingir esse objetivo, alguns recursos foram empregados e torna-se importante deixá-los claros.

2 Estruturas Fundamentais de Indexação e Busca

Para que fosse possível buscar as informações dos documentos de uma forma efetiva e rápida, em termos computacionais, foi implementada uma árvore `Trie` que armazena as informações de todos os arquivos e de todas as palavras presentes em todos os artigos. Além disso, utilizou-se o índice reverso para localizar aonde a palavra aparece no Corpus.

2.1 1. Construção do Índice Reverso com `TrieCompacta`

Para a construção do índice reverso, foi necessário implementar uma árvore `Trie`. Para isso, foi implementado uma classe `TrieCompacta`, porque optamos por utilizar a versão compacta para melhor uso de memória. Algo interessante é que, estudando como se faz índices invertidos, aprendemos que também é feito usando `hashmaps`, mas o uso de uma `TrieCompacta` é interessante porque permite o *autocomplete* e erros leves de digitação, usando métricas como Distância de Levenstein para buscar elementos.

- **A Estrutura do Nó (`Node`):** Dentro da `Trie`, temos uma classe `Node`, que representa os nós da árvore. Esse nó tem um valor, tem filhos, tem uma *flag* se é o nó terminal ou não (algo que fizemos nos slides da disciplina com o '$') e armazena em um vetor `positions` aonde no Corpus apareceu aquele prefixo.
- **O Vetor `positions` (Localização no Corpus):** O vetor de `positions` é interessante porque ele tem o formato de um vetor de objetos, cada objeto tem um `path` que referencia em qual arquivo aquele prefixo acontece, e um vetor `locations` que marca dentro daquele arquivo, em ordem crescente, aonde aquele prefixo acontece. Isso é importante para calcular o *z-score* para a ordenação de quais são os arquivos mais pertinentes.

2.2 2. Desafio na Marcação de Localização (`locations`)

Uma das dificuldades encontradas na hora de marcar no vetor `locations` aonde naquele arquivo o prefixo acontece foi que, no início, estávamos quebrando o texto em espaços () e procurando as palavras ali. O problema é que isso não nos permitia saber de forma rápida o *offset* de onde aquela palavra acontece no Texto. Então, começamos a usar o `find` com uma *flag* muito interessante de `start_pos`, para começar a buscar a partir de uma posição, que faz a consulta ser mais rápida. Para fazer o *z-score*, calculamos inicialmente a média de ocorrências do termo por documento (`avg_occurrence`) dividindo o número total de ocorrências pelo número de documentos na coleção. Em seguida, calculamos o desvio padrão (`std_occurrence`) das ocorrências usando a fórmula estatística tradicional: a raiz quadrada da média dos quadrados das diferenças entre cada contagem de ocorrências e a média. Finalmente, o *z-score* de cada documento é calculado como a diferença entre sua contagem de ocorrências e a média, dividida pelo desvio padrão. Quando o desvio padrão é zero (todos os documentos têm a mesma contagem), usamos simplesmente a diferença em relação à média. Esta abordagem nos permite ranquear os documentos com base na significância estatística da frequência do termo, priorizando documentos onde o termo aparece com frequência anormalmente alta em relação à distribuição geral na coleção.

3 Avaliação de Consultas Booleanas: A AST

Ademais, é necessário também falar sobre como foi resolvida a avaliação das consultas booleanas. Para isso, foi implementada uma árvore de sintaxe abstrata (`AST`), construída a partir de um `parser`, que em conjunto com seu avaliador, torna possível recuperar as informações de quais arquivos atendem aos critérios de uma determinada consulta.

Para a avaliação da consulta booleana, foram implementadas quatro classes necessárias para seu funcionamento: `ASTnode`, `AST`, `QueryParser` e `ASTEvaluator`.

3.1 1. A Estrutura da Árvore: `ASTnode` e `AST`

As duas primeiras classes, `ASTnode` e `AST`, são as classes que possibilitam a construção da árvore de sintaxe binária e a primeira funciona como os nós da árvore que é gerenciada pela segunda.

- `ASTnode`**:** Define os componentes de um nó da `AST`, como o tipo do nó (`TERM` ou `OPERATOR`), o valor, o filho à direita, o filho à esquerda e o pai.
- `AST`**:** Funciona como construtora e gerenciadora da árvore de sintaxe binária, guardando a raiz e permitindo a caminhada.

3.2 2. Construção da AST com `QueryParser`

Essa classe é extremamente importante para a construção correta da árvore. Ela funciona como o avaliador da *string* de entrada correspondente à consulta lógica, transformando-a em uma `AST` ao considerar a ordem de precedência dos operadores e a existência de parênteses. Ela utiliza quatro métodos principais, chamados recursivamente para simular uma gramática: `parse_expression`, `parse_or_expression`, `parse_and_expression` e `parse_primary`.

O processo inicia com o método `parse_expression`, que chama `parse_or_expression`. Este, por sua vez, tenta construir nós para o operador `OR` de forma recursiva, chamando `parse_and_expression` para seus operandos. O método `parse_and_expression` faz o mesmo para o operador `AND`, mas delegando a resolução de termos ou expressões entre parênteses ao método `parse_primary`. Finalmente,

`parse_primary` é responsável por identificar e consumir os tokens da consulta: se encontrar um parêntese aberto '(', inicia uma nova expressão; se encontrar uma palavra, cria um nó do tipo `TERM`; caso contrário, lança um erro de sintaxe. Essa hierarquia de métodos garante que a precedência dos operadores (`AND` antes de `OR`) e o agrupamento por parênteses sejam corretamente respeitados na estrutura da árvore gerada.

3.3 3. Execução da Consulta com `ASTEvaluator`

Por fim, a avaliação é feita através da classe `ASTEvaluator`, que caminha pelos nós da `AST` (das folhas até a raiz), encontrando os documentos que:

- Possuem o termo (`TERM`).
- Atendem ao operador `AND` (termos presentes em ambos os filhos).
- Atendem ao operador `OR` (termos presentes em pelo menos um dos filhos).

Assim, a avaliação da consulta lógica é concluída, retornando o conjunto de documentos que atendem aos critérios da busca feita pelo usuário.

3.4 4. A Interface: `TrieAdapter`

Por fim, é necessário comentar que foi preciso implementar uma classe para fazer a interface entre a árvore `Trie` que implementamos para armazenar as informações das palavras e dos arquivos e a `AST` que foi implementada para possibilitar a avaliação das consultas booleanas, feita através da classe `TrieAdapter`. Essa classe tem como objetivo mapear os caminhos de arquivo (*strings*) para identificadores numéricos únicos (`doc_id`) e vice-versa, criando uma camada de abstração que facilita a manipulação dos documentos durante a avaliação. Ela também adapta o método de consulta da `Trie` (`find`) para um formato compatível com o `ASTEvaluator`, traduzindo a lista de ocorrências, que originalmente associa um termo aos caminhos dos arquivos e suas posições, para uma lista de tuplas contendo o `doc_id` e as posições correspondentes. Dessa forma, o `ASTEvaluator` pode operar exclusivamente com identificadores numéricos, simplificando as operações de interseção (`AND`) e união (`OR`) durante a avaliação da consulta.

4 Arquitetura e Experiência do Usuário (UX)

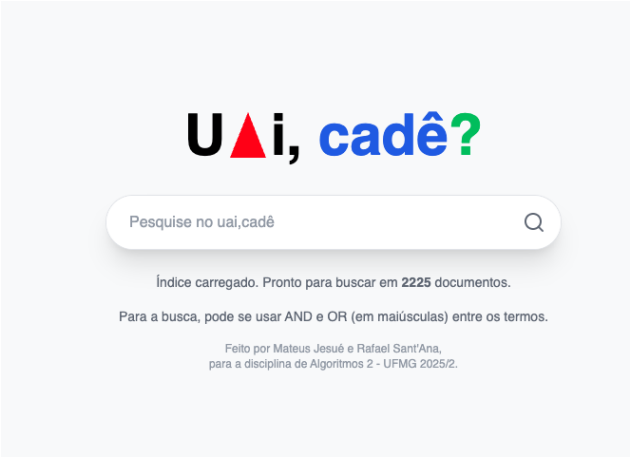
4.1 1. Frontend com `Flask`

Sobre o `Frontend` da aplicação, foi construído usando `Flask` e `HTML Templates` uma interface que se assemelha um pouco a como o `Google` é, mas com uma identidade visual inspirada no `Cadê`, famoso buscador do Brasil de 1995.

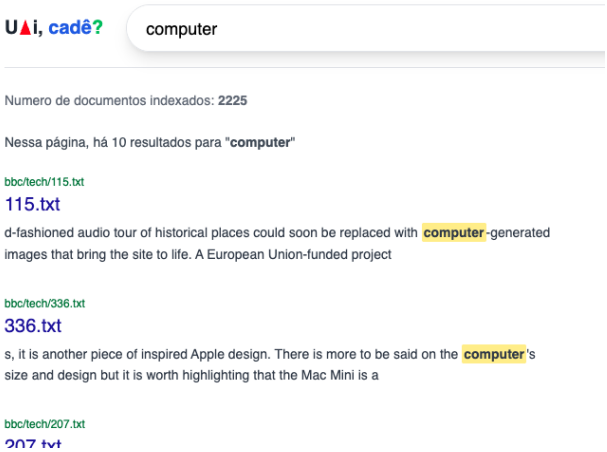
- Quando se faz uma consulta, o trabalho do `frontend` é pedir ao `backend` para que retorne um `HTML` com as páginas mais pertinentes para aquela consulta.
- Foi implementado um sistema de paginação das páginas mais pertinentes.
- Foi implementada também uma tela para visualizar o documento em questão.

4.2 2. Backend de Processamento e *Highlighting*

Sobre o `Backend` de processamento de consulta, assumimos que uma consulta é simples dentro de cada um dos operadores, isso é, consultamos apenas uma palavra por vez. Além disso, para



(a) Tela inicial de Busca do Cadê



(b) Consulta do termo 'Computer'

simplificar a interpolação de elementos ao `frontend`, foi computado no `backend` mesmo algumas *tags* `html` para *highlighting* de onde o termo procurado apareceu.

5 Conclusão

O projeto do motor de busca booleano foi um exercício complexo que combinou o uso de estruturas de dados eficientes, como a `Trie` compacta para indexação e a `AST` para avaliação de consultas lógicas. A correta interface entre essas estruturas e o desenvolvimento de um `frontend` interativo com `Flask` possibilitaram a criação de um sistema funcional que atende aos objetivos propostos.

6 Referências

- Slides da disciplina de Algoritmos II
- Cormen, Thomas H. Algoritmos: Teoria e Prática, Editora Campus, v. 2 p. 296, 2002