Trabalho Prático 1 – Manipulação de sequências: Trie-legal

Leonardo César Cota de Castro, 2023087923

Luis Felipe Pimenta Marcos, 2023087907

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

---

1. Introdução

Este relatório detalha a implementação do Trie-legal, um protótipo de máquina de busca desenvolvido para o Trabalho Prático 1 da disciplina de Algoritmos 2. O objetivo principal do trabalho era aplicar os conceitos de árvores de prefixo na construção de um índice invertido e, com isso, desenvolver uma aplicação web de busca funcional.

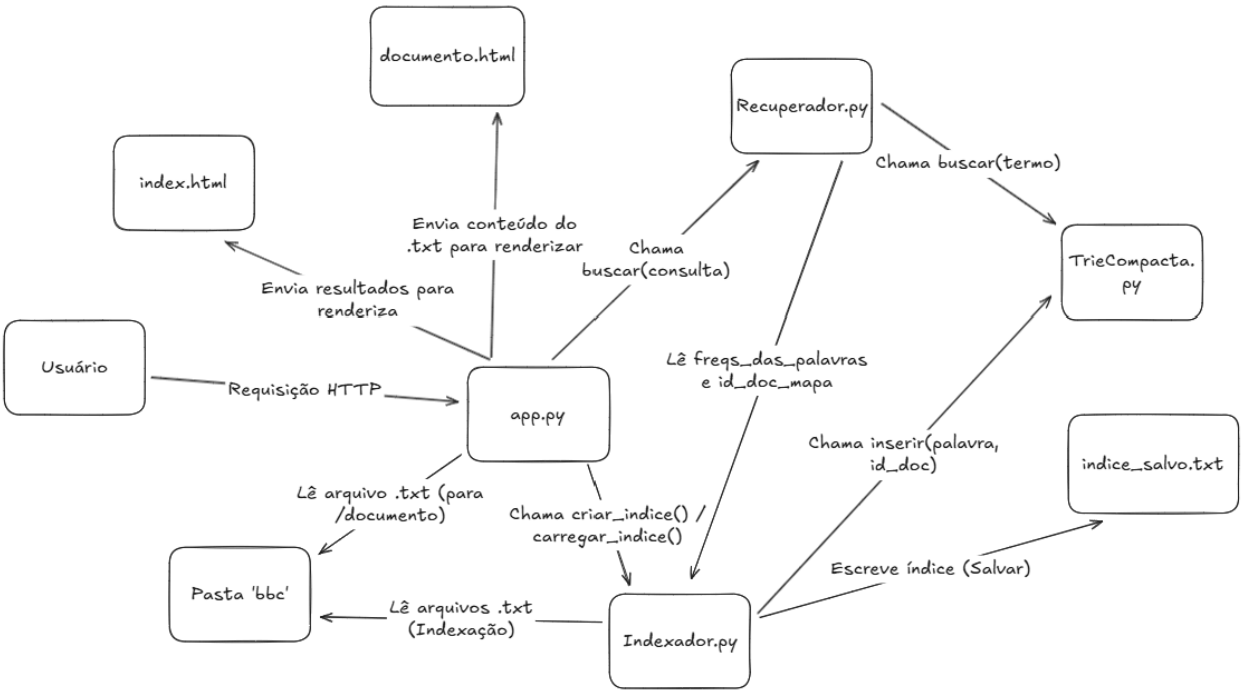
O sistema desenvolvido utiliza a estrutura Trie Compacta para criar um índice invertido sobre um corpus de 2225 notícias da BBC News, categorizadas em cinco áreas temáticas. O módulo de busca é baseado em um modelo de consulta Booleana e os resultados são ordenados por um sistema de ranking baseado no z-score. A interface com o usuário foi desenvolvida em Flask e é responsável pela exibição paginada dos resultados.

2. Arquitetura da solução

O sistema é modularizado em três componentes principais de lógica e um de interface, conforme a estrutura abaixo:

1. Módulo de Indexação (Indexador.py): Responsável pela leitura, processamento e tokenização dos documentos, e pela construção e persistência do índice invertido em disco.
2. Módulo de Estrutura de Dados (TrieCompacta.py): Implementa a estrutura da árvore Trie compacta do zero, utilizada para armazenar a associação entre termos e documentos (doc_id).
3. Módulo de Recuperação (Recuperador.py): Responsável por analisar a consulta, executar a lógica Booleana sobre os conjuntos de documentos, calcular a relevância (z-score) e gerar os snippets.
4. Interface Web (app.py, index.html, documento.html): Baseada em Flask, gerencia o servidor, processa as requisições de busca, realiza a paginação e exibe os resultados e o conteúdo completo dos documentos.

Fluxo de Execução: O servidor app.py carrega ou cria o índice ao iniciar. Ao receber uma consulta via /search, o Recuperador utiliza a Trie para obter os documentos, calcula os z-scores, ordena os resultados e os retorna ao app.py, que renderiza o index.html com a paginação de 10 resultados.



3. TrieCompacta.py

Desenvolvimento total da Trie Compacta, que é a base do índice invertido. Diferente de uma árvore comum que guarda letra por letra, esta estrutura comprime os galhos, guardando pedaços de palavras em cada nó. Ela é responsável por guardar as palavras que aparecem em quais documentos e faz isso de uma forma que economiza muita memória.

class NoTrie: Esta é a classe que representa cada nó da árvore. Seus atributos principais são:

prefixo: Armazena o fragmento de palavra que este nó representa.

filhos: Uma lista de outros objetos NoTrie que são descendentes deste.

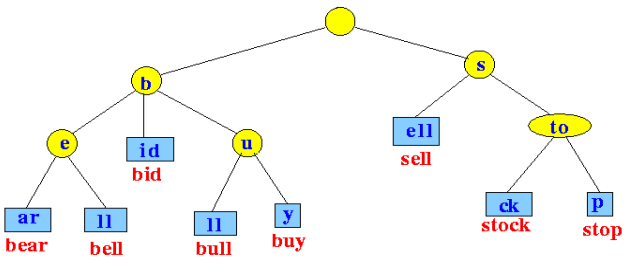
noticias: Um conjunto que armazena os IDs dos documentos onde a palavra exata que termina neste nó é encontrada.

class TrieCompacta: Esta é a classe que gerencia a árvore como um todo. As funções são:

inserir(self, palavra, id_noticia): Esta função é responsável por adicionar uma nova palavra e o ID do documento associado. Ao invés de simplesmente adicionar um nó, ela precisa lidar com quatro casos lógicos para manter a árvore compactada:

1. Caso 1 (Continuação): A palavra a ser inserida contém o prefixo do nó filho. A função continua a inserção recursivamente no filho.
2. Caso 2 (Quebra no Nó): A palavra a ser inserida é um prefixo de um nó filho. Vai dividindo as palavras, a depender da possibilidade de compactar ou não.
3. Caso 3 (Idênticos): A palavra já existe. Apenas adicionamos o id_noticia ao set noticias do nó.
4. Caso 4 (Quebra no Meio): O caso mais complexo, onde a palavra e um filho compartilham um prefixo parcial. Um novo nó é criado, e ele passa a ter dois filhos, com os dados da nova palavra e com os dados do nó antigo.

buscar(self, palavra): Esta função navega pela árvore compactada para encontrar uma palavra exata. Se encontrar, retorna os IDs de documentos associados a ela, ou retorna um set vazio.



4. Indexador.py

Este arquivo é o construtor do nosso índice. Sua função é ler todos os 2225 arquivos de texto do corpus da BBC, extrair cada palavra e alimentar a TrieCompacta. Além de construir o índice em memória , ele também é responsável por salvar esse índice pronto em um arquivo de formato .txt e por carregá-lo rapidamente na próxima vez que o programa iniciar, conforme exigido.

criar_indice(self): Varre todas as categorias de bbc. Para cada arquivo .txt encontrado, ela atribui um ID numérico único, que é armazenado no id_doc_mapa, e chama a função _processar_arquivo.

_processar_arquivo(self, caminho_arquivo, id_doc): Para cada arquivo: Lê o texto, converte para minúsculas e usa uma expressão regular (re.findall) para extrair apenas as palavras.

Para cada palavra, ele faz duas coisas:

1. self.trie.inserir(palavra, id_doc): Insere a palavra na Trie, associando-a ao ID do documento. Isso constrói o índice para a busca booleana.
2. self.freqs_das_palavras[...]: Atualiza um dicionário que conta a frequência ({palavra: {doc_id: freq}}). Este mapa é essencial para o cálculo de relevância (z-score) exigido.

salvar_indice(self) e carregar_indice(self): Conforme solicitado, implementamos funções para persistir o índice em disco. O arquivo salvo tem duas seções: um mapa de documentos (doc:ID:caminho_do_arquivo) e o índice de frequências (palavra:id1,freq1;id2,freq2;...). Optamos por um formato de texto por ser mais fácil de depurar e inspecionar manualmente, embora tenha o argumento contrário de que um formato binário pudesse resultar em um arquivo de índice mais compacto.

carregar_indice: lê o arquivo de volta para a memória, populando o id_doc_mapa, o freqs_das_palavras e, crucialmente, reconstruindo a TrieCompacta em memória para que o servidor possa iniciar rapidamente.

5. Recuperador.py

Este código é o mecanismo de busca propriamente dito. Ele conecta a TrieCompacta com a lógica de busca do usuário, interpreta os operadores booleanos AND e OR , e encontra os documentos correspondentes. Também é responsável por calcular o z-score de cada documento, ordená-los do melhor para o pior e gerar o snippet com o termo destacado para ser mostrado na página de resultados.

buscar(self, consulta): Função principal chamada pelo servidor. Ela coordena todo o processo: processa a consulta, avalia a expressão booleana , calcula a relevância e formata os resultados.

_avaliar_expressao(self, tokens): Esta função implementa o modelo booleano.

1. Primeiro, ela usa a trie.buscar() para obter o set de documentos para cada termo puro da consulta.
2. Depois, ela traduz a consulta do usuário para uma expressão Python válida. Os operadores AND e OR (em maiúsculas} são convertidos para os operadores de conjunto & e |.
3. Ela então usa eval() para executar a operação lógica entre os conjuntos (ex: eval("set_A & (set_B | set_C)"))

_calcular_relevancia_docs(self, docs, palavras): Esta é a implementação do ranking por z-score. Para ordenar os resultados por relevância:

1. Para cada termo da consulta, calculamos sua média de frequência e desvio padrão (pstdev) em toda a coleção (usando o mapa freqs_das_palavras e o total_docs).
2. Para cada documento retornado pela busca booleana, calculamos o z-score de cada termo da consulta: z = (frequencia_no_doc - media_geral) / desvio_padrao.
3. A relevância final do documento é a média dos z-scores de todos os termos da consulta.

_gerar_snippet(self, caminho_arquivo, palavra): Para o front-end, esta função gera o trecho de exibição.

1. Ela primeiro identifica qual dos termos da consulta é o mais frequente naquele documento específico.
2. Ela abre o arquivo original, encontra a primeira ocorrência desse termo e captura 80 caracteres antes e 80 depois.
3. Finalmente, ela envolve o termo encontrado com as tags ... para que o CSS possa destacá-lo em amarelo no front-end.

6. app.py

O app.py é o servidor Flask que conecta tudo. Ele é responsável por iniciar o sistema, carregar o índice - ou criá-lo, se for a primeira vez - e expor as rotas que o usuário acessa. Ele cuida de receber a requisição do navegador, passar a consulta para o Recuperador, receber a lista de resultados, aplicar a paginação (mostrando 10 por página) e, por fim, renderizar os templates HTML com os dados para o usuário.

Inicialização: O código principal do app.py é executado uma vez no início. Ele instancia o Indexador e verifica se o índice existe. Se sim, ele carrega (carregar_indice), se não, ele cria (criar_indice) e salva. Isso garante que o servidor esteja sempre pronto.

@app.route('/') (Home): Apenas exibe a página de busca (index.html).

@app.route('/search'): É a rota principal. Ela recebe a consulta do usuário, chama o recuperador_global.buscar(), mede o tempo da busca e faz a paginação (exibindo apenas 10 resultados de cada vez).

@app.route('/documento/<int:doc_id>'): É a página que exibe um documento. Ela usa o doc_id para encontrar o caminho do arquivo .txt original, lê seu conteúdo e o envia para o template documento.html.

7. index.html

Usa a sintaxe do Jinja2 para exibir dinamicamente a lista de resultados. Responsável pela configuração da home principal do projeto.

Visual: Ele tem um vídeo (wallpaper.mp4) rodando em loop no fundo da tela.

Busca: Mostra uma barra de pesquisa no topo, que "lembra" o que o usuário digitou anteriormente.

Estatísticas: Logo abaixo da busca, exibe um resumo (ex: "Página 1 de 10", "Encontrados 100 resultados em 0.0005s").

Resultados: É a parte principal. Ele faz um loop e cria um item para cada resultado, mostrando seu título, que é um link, um snippet - um trecho do documento, com o termo buscado - e o score de relevância.

Paginação: Se houver muitas páginas de resultados, ele mostra os links de "Anterior" e "Próxima" no final.

8. documento.html

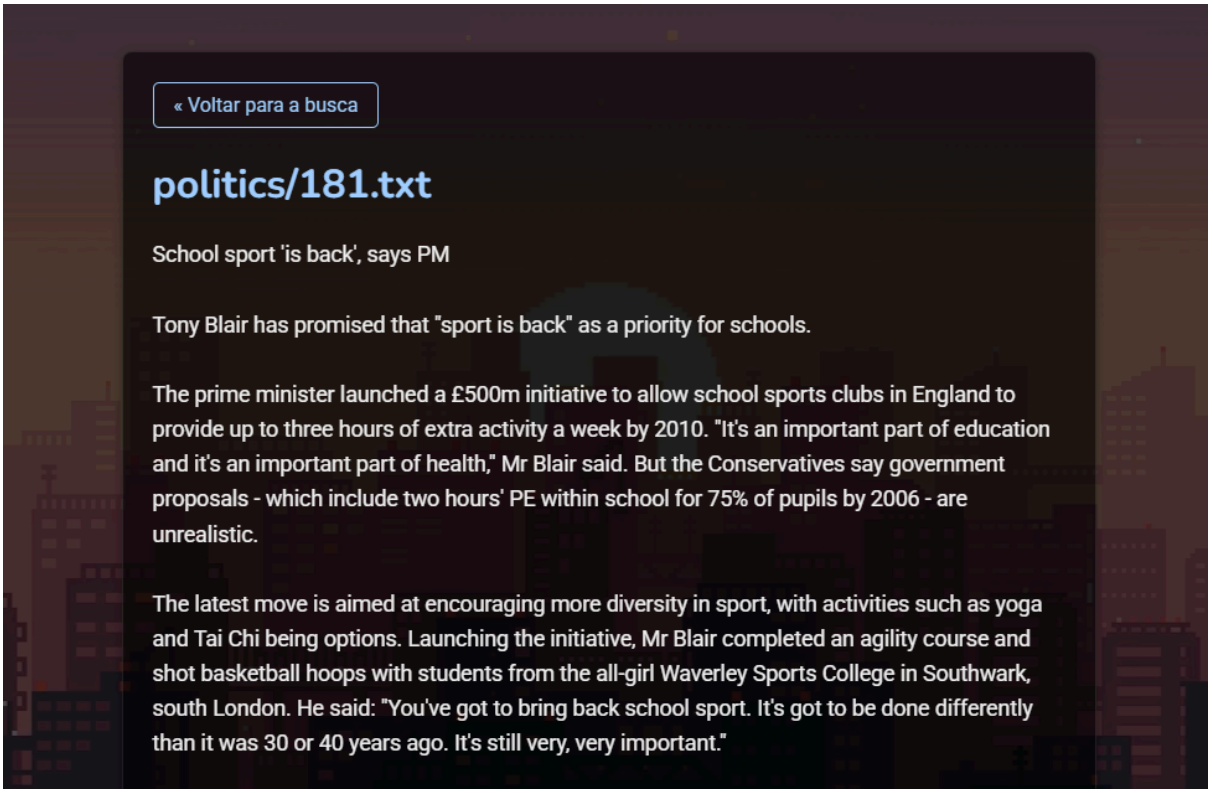
Este é o template HTML para a página de visualização de um documento específico. É para onde o usuário vai depois de clicar em um resultado da busca.

Visual: Ele reutiliza o mesmo fundo de vídeo da página de busca.

Navegação: Tem um link "Voltar para a busca" que simplesmente usa o histórico do navegador para retornar à lista de resultados.

Conteúdo: A parte principal exibe o titulo do documento e o seu conteudo.

Formatação: O conteúdo é exibido dentro de uma tag <pre>, o que é importante: isso preserva toda a formatação original do texto, incluindo espaços em branco e quebras de linha (muito usado para mostrar código ou documentos formais).



9. main.css

Este é o arquivo CSS principal que dá a identidade ao projeto Trie-legal, foi criado para melhorar no critério de qualidade visual. Ele define toda a aparência das duas páginas HTML especificadas anteriormente.

Tema Visual: É um design mais limpo. Ele fixa um wallpaper no fundo da tela (.video-background) e coloca um filtro escuro semi-transparente por cima (.video-overlay). Isso faz com que o texto, que é branco, tenha boa legibilidade.

Fontes: Define Roboto como a fonte padrão e usa Nunito, uma fonte mais pesada e arredondada, para o título principal Trie-legal (.main-title).

Layout: Centraliza todo o conteúdo principal (.content) em uma coluna no meio da tela, com largura máxima de 800px.

Barra de Busca: Estiliza o formulário de busca como uma pílula, com o campo de texto tendo cantos arredondados à esquerda e o botão à direita.

Cartões de Resultado: Define o estilo dos itens de resultado (.result-item) e do contêiner do documento (.document-container) como cartões com fundo preto semi-transparente, bordas arredondadas e uma leve sombra.

Snippet: A classe .result-snippet strong é a que define o marca-texto amarelo para os termos encontrados na busca.

Interatividade: A regra .result-link é importante: ela remove o sublinhado padrão dos links e faz com que o cartão de resultado inteiro seja clicável, escurecendo levemente quando o mouse passa por cima.

Formatação do Documento: Na página do documento, a regra .document-content garante que o texto preserve as quebras de linha e espaços (pre-wrap), para que a formatação original seja mantida.

10. Conclusão

A realização deste trabalho prático permitiu abordar de forma prática os algoritmos de manipulação de sequências vistos em aula, com foco específico na implementação de uma árvore de prefixo para a construção de um índice invertido. Além disso, o projeto cumpriu seu objetivo secundário ao nos dar a oportunidade de ir além dos tópicos centrais da disciplina, exigindo o aprendizado de conceitos relacionados à recuperação de informação, ao funcionamento de máquinas de busca e ao uso de bibliotecas para construção de aplicações web, como o Flask.

A estrutura Trie Compacta foi implementada e corrigida para garantir o funcionamento correto da indexação e da busca. O sistema de busca Booleana foi integrado ao ranking de relevância por z-score , e a solução de front-end provê uma interface funcional, paginada e com a capacidade de visualizar o conteúdo completo dos documentos. O índice garante a persistência dos dados e a conformidade com as restrições impostas.