Relatório — Trabalho Prático 1
Implementação de uma Máquina de Busca com Trie Compacta

Bruna Oliveira Luna Almeida
Universidade Federal de Minas Gerais (UFMG)
Disciplina: DCC207 — Algoritmos e Estruturas de Dados II
brunaluna@ufmg.br

Outubro de 2025

Resumo

Este trabalho descreve o desenvolvimento de uma máquina de busca compacta baseada em uma Trie, capaz de indexar e consultar documentos do corpus *BBC News*. O sistema integra uma API em Flask e uma interface web denominada **Bubble**, que oferece funcionalidades de autocomplete, paginação e destaque de termos. O objetivo principal foi aplicar os conhecimentos de árvores de prefixo e índices invertidos em um contexto prático, aproximando o conteúdo teórico das aulas a uma aplicação real.

1 Introdução

O trabalho propõe a construção de uma máquina de busca compacta sobre o corpus *BBC News*. A solução integra uma API desenvolvida em Flask, responsável pela indexação e consulta dos documentos, a uma interface web ape-

lidada de **Bubble**, que oferece uma experiência interativa de busca com autocompletar, paginação e destaque de termos.

O objetivo central foi aplicar conceitos de estruturas de dados — especialmente árvores de prefixo — à construção de índices invertidos, consolidando o conteúdo ensinado em sala de aula e demonstrando sua aplicabilidade prática. O projeto também encoraja o uso de tecnologias não tradicionais no contexto da disciplina, como Flask, Angular e Three.js.

O sistema foi dividido em três módulos principais: (i) o módulo de estrutura de dados, responsável pela implementação da Trie compacta; (ii) o módulo de indexação, encarregado de processar os documentos e construir o índice invertido; e (iii) o módulo de recuperação, responsável por interpretar consultas booleanas e calcular a relevância dos documentos. A aplicação web fornece uma interface simples e eficiente, aproximando o comportamento do sistema ao de um mecanismo de busca tradicional.

2 Metodologia

A metodologia adotada seguiu uma abordagem modular, na qual cada componente foi desenvolvido e testado de forma independente antes da integração final.

Primeiramente, foi projetada a estrutura Trie compacta, uma árvore de prefixos otimizada para reduzir redundâncias. Ela compartilha prefixos comuns entre palavras, armazenando nós comprimidos que permitem buscas e autocompletar de maneira eficiente.

Na etapa seguinte, foi construído o módulo de indexação, responsável por percorrer o corpus de documentos, tokenizar o conteúdo e inserir os termos obtidos na Trie compacta. Cada nó terminal da estrutura passou a armazenar os documentos e suas respectivas frequências, compondo o índice invertido. Após a indexação, foram calculadas estatísticas descritivas (média e desvio padrão das frequências) para permitir a normalização das relevâncias via *z-score*.

O módulo de recuperação de informação interpreta as consultas textu-

ais e suas expressões booleanas (AND, OR e parênteses), incluindo operadores implícitos. A partir dos termos buscados, o sistema combina os conjuntos de documentos e calcula a média dos *z-scores* para definir a ordem de relevância.

Embora um dicionário auxiliar seja mantido em memória para fins estatísticos, a Trie compacta é a estrutura central utilizada durante a busca, armazenando referências diretas aos documentos.

Por fim, o sistema foi integrado a uma interface web desenvolvida com Flask e Angular, que apresenta os resultados de forma dinâmica e interativa. O backend Flask conecta os módulos de indexação e recuperação ao frontend, permitindo ao usuário realizar buscas de forma fluida.

3 Decisões de Projeto

Diversas decisões de projeto foram tomadas para garantir clareza estrutural e eficiência.

A leitura do corpus foi feita diretamente via `ZipFile`, reduzindo o consumo de disco. Um dicionário Python foi usado inicialmente para validar o comportamento do índice invertido, mas foi posteriormente incorporado à Trie compacta. Assim, cada nó terminal passou a armazenar os documentos e suas frequências, permitindo operações de interseção e união rápidas — essenciais para consultas booleanas.

Além disso, cada termo possui os parâmetros estatísticos de média e desvio padrão, viabilizando o cálculo de relevância normalizada. Essa técnica impede que termos extremamente comuns dominem o ranking de resultados.

Para o processamento das consultas, adotou-se uma abordagem baseada em pilhas, garantindo a precedência correta entre operadores e o suporte a expressões aninhadas. O cálculo de relevância utiliza uma média logarítmica dos *z-scores*, suavizando diferenças extremas.

No front-end, priorizou-se leveza e compatibilidade. O arquivo `index.html` carrega um único bundle de JavaScript, contendo estilos e scripts do Angular, além de animações criadas com Three.js. Essa decisão simplifica o carregamento

e torna a aplicação mais responsiva. O servidor Flask também habilita CORS, permitindo que futuros clientes hospedados em outros domínios acessem a API.

4 Implementação

O arquivo `app.py` é o ponto de partida da aplicação. Ele inicia o servidor Flask, habilita a comunicação com o navegador e define as rotas que conectam a interface ao backend. Durante a execução, o sistema realiza a indexação inicial dos documentos — limitada a duzentos arquivos por padrão, podendo ser ajustada conforme necessário.

A classe `Indexador`, localizada em `core/indexer.py`, contabiliza as ocorrências das palavras, calcula estatísticas e interage com a Trie, além de oferecer funções para salvar e carregar o índice. O módulo `core/retriever.py` interpreta as consultas, combina os resultados e calcula a relevância. Já a Trie compacta, em `core/trie.py`, organiza os termos de forma eficiente, permitindo buscas rápidas e sugestões automáticas.

O arquivo `indice_invertido.txt`, em formato JSON, armazena o índice criado durante a indexação e permite seu carregamento em execuções futuras, sem reprocessar o corpus. As dependências do projeto estão listadas em `requirements.txt` e foram testadas em um ambiente Python 3.12 configurado na pasta virtual `.venv`.

5 Exemplos de Uso

Para verificar o funcionamento do sistema, foram realizados diversos testes com o corpus da BBC. Inicialmente, após a indexação, a Trie compacta foi inspecionada por meio de inserções de teste, confirmando que prefixos comuns eram devidamente agrupados. Por exemplo, a inserção das palavras *batata* e *batalha* resultou em um nó compartilhado com prefixo “bata”, demonstrando a eficácia da compactação. Consultas de autocompletar, como “ba”, retornaram corretamente sugestões de termos como *batata* e *batalha*, validando o funcionamento do

método `sugestoes`.

Exemplo I — `(economy AND government)`

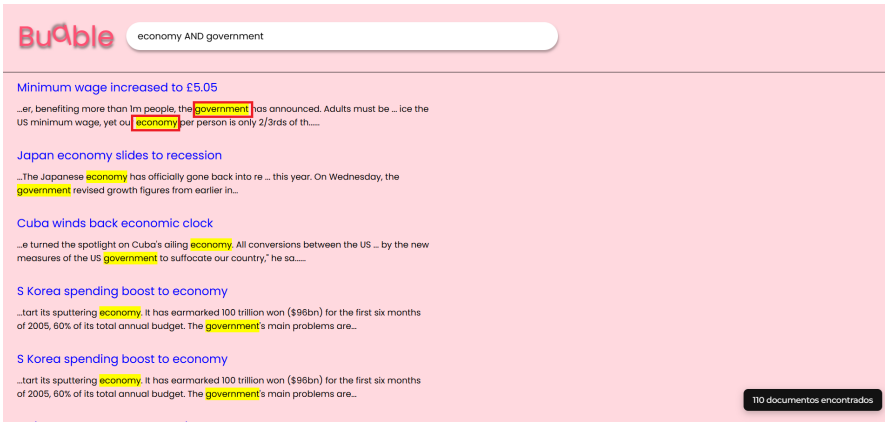

Figura 1: Consulta `(economy AND government)` mostrando termos destacados no texto.

Exemplo II — `(sport OR technology)`

Figura 2: Consulta `(sport OR technology)` exibindo documentos combinados.

Exemplo III — `(economy AND government) OR sports`

Figura 3: Consulta composta `(economy AND government) OR sports`.

Exemplo IV — `finance market`

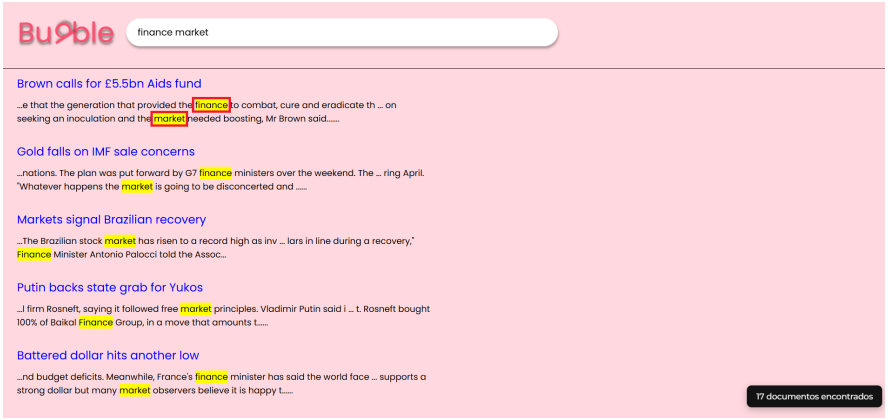

Figura 4: Busca por múltiplos termos simples: `finance market`.

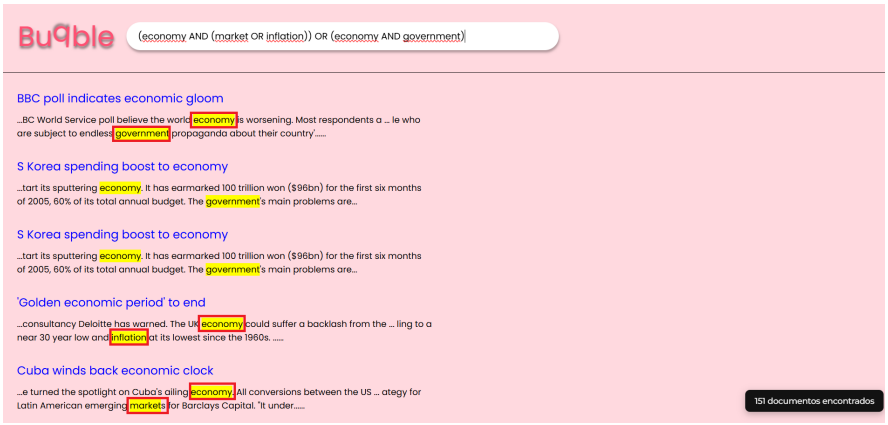

Figura 5: Consulta complexa aninhada: `(economy AND (market OR inflation)) OR (economy AND government)`.

Exemplo V — `(economy AND (market OR inflation)) OR (economy AND government)`

Exemplo VI — Sugestões de Autocomplete

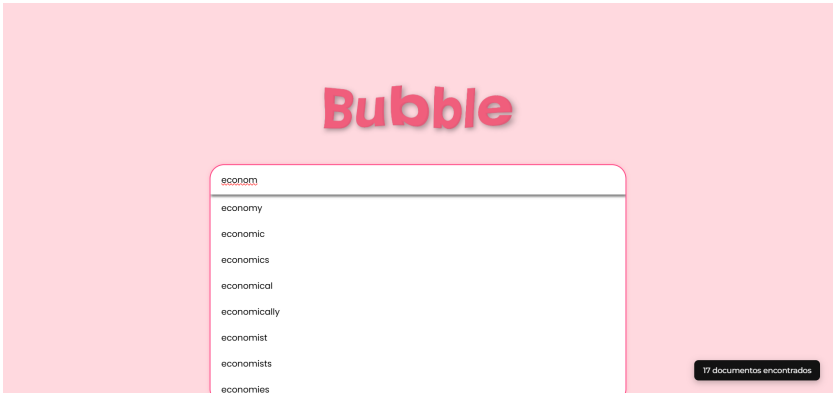

Figura 6: Primeiro termo de busca exibindo sugestões automáticas.

Os resultados foram ordenados por relevância, exibindo em primeiro lugar os textos com maior densidade relativa dos termos pesquisados. No navegador, a interface de busca permitiu digitar consultas e visualizar os resultados paginados,

Figura 7: Segundo termo de busca com sugestões geradas pela Trie compacta.

com dez documentos por página. Cada resultado apresentava um trecho de 160 caracteres do texto original, contendo o termo de busca destacado. Essa funcionalidade reproduz a experiência típica de um mecanismo de busca, permitindo ao usuário avaliar rapidamente o contexto de cada ocorrência.

6 Conclusões

O desenvolvimento do Trabalho Prático 1 possibilitou compreender, de forma aplicada, como estruturas clássicas de dados podem ser utilizadas em sistemas reais de recuperação de informação. A Trie compacta demonstrou eficiência na representação de strings e na redução de redundâncias, enquanto o índice invertido e o cálculo de *z-score* mostraram-se adequados para definir a relevância dos documentos.

Em suma, o trabalho alcançou seus objetivos de aplicar conceitos teóricos à construção de uma aplicação completa, consolidando o aprendizado sobre árvores de prefixo e sistemas de busca, além de promover a integração entre backend e frontend em um ambiente moderno.

Referências

- Radix tree. Disponível em: https://en.wikipedia.org/wiki/Radix_tree.
- JAMIEGO. *Data Structures in Golang — The Trie Data Structure*. Disponível em: https://www.youtube.com/watch?v=H-6-8_p88r0.
- INSIDE CODE. *Trie Data Structure*. Disponível em: https://www.youtube.com/watch?v=qA8l8TAMyig.
- TUTORIALS POINT. *Compressed Tries*. Disponível em: https://www.tutorialspoint.com/data_structures_algorithms/compressed_tries.htm.
- COMPUTERBREAD. *Compressed Trie*. Disponível em: https://www.youtube.com/watch?v=qakGXuOW1S8.