Trabalho Prático 1: Desenvolvimento de um Sistema de Busca com Trie Compacta

Paula D'Agostini Alvares Maciel
pauladagostini@ufmg.br

Pedro Bernardo Goulart Parreira
pedrobgoulart@ufmg.br

Outubro de 2025

Abstract

Este relatório descreve o desenvolvimento de um protótipo de sistema de busca, criado para a disciplina de Algoritmos 2 (DCC207). O projeto implementa um processo completo de recuperação de informação, que inclui a indexação de documentos, o processamento de buscas com operadores booleanos (AND/OR) e a ordenação dos resultados por relevância. A principal estrutura de dados utilizada é um índice invertido baseado em uma Árvore Trie Compacta, que foi totalmente implementada por nós. A relevância é calculada usando a média dos z-scores dos termos pesquisados, e a interface com o usuário foi desenvolvida como uma aplicação web com o framework Flask. Recursos adicionais como persistência de índice, geração de snippets de texto e paginação também foram implementados.

1 Introdução

O objetivo deste trabalho foi aplicar os conceitos de algoritmos de manipulação de sequências na construção de uma máquina de busca funcional. O sistema foi projetado para indexar uma coleção de documentos e permitir que os usuários encontrem informações relevantes através de consultas.

Utilizamos o corpus de notícias da BBC News, de Greene e Cunningham (2006), que contém 2225 textos. O projeto foi desenvolvido em Python, com a interface web criada em Flask, seguindo os requisitos do trabalho. Este documento apresenta a arquitetura da solução, as estruturas de dados, os algoritmos e as decisões de projeto tomadas ao longo do desenvolvimento.

2 Arquitetura e Metodologia

Projetamos a solução de forma modular para organizar o código e facilitar os testes. Cada parte principal do sistema está em seu próprio arquivo dentro da pasta `src/`.

2.1 Estrutura de Arquivos do Projeto

A organização final do projeto ficou da seguinte forma:

```
TP1-ALG2/
|-- main.py                # Orquestrador da aplicacao Flask
|-- bbc-fulltext.zip       # Corpus de documentos
|-- my_corpus.idx          # Arquivo do indice salvo
|-- README.md
|-- src/
|   |-- __init__.py
|   |-- insert.py          # Logica da Trie Compacta
|   |-- indexing.py        # Funcoes de indexacao e persistencia
|   |-- search.py          # Logica da busca booleana
|   |-- relevance.py       # Calculo do z-score e ranking
|   `-- utils.py           # Funcoes auxiliares (snippet)
|-- static/
|   `-- style.css          # Estilos da interface
```

```
|-- templates/
|   |-- index.html          # Pagina inicial
|   `-- results.html        # Pagina de resultados
`-- venv/                     # Ambiente virtual Python
```

2.2 Ferramentas e Metodologia

O desenvolvimento foi realizado utilizando a linguagem Python e o framework Flask. Para a escrita deste relatório, utilizamos a plataforma Overleaf com LaTeX. Durante todo o processo, o código foi versionado com **Git** e gerenciado em um repositório no **GitHub**, o que permitiu um trabalho colaborativo organizado e um histórico completo das alterações.

3 Estrutura de Dados: Trie Compacta

A base do nosso sistema de busca é um índice invertido, que implementamos usando uma Árvore Trie Compacta (Radix Tree). Essa estrutura é eficiente para buscar palavras, e a compactação ajuda a economizar memória. A implementação está no arquivo `src/insert.py`.

Cada nó da árvore é uma `dataclass` com dois campos principais:

- **postings:** Um dicionário que guarda os arquivos onde a palavra (formada pelo caminho até aquele nó) aparece. As chaves são os nomes dos arquivos e os valores são a frequência da palavra naquele arquivo. Essa contagem de frequência é fundamental para o cálculo de relevância.
- **branches:** Um dicionário que guarda os "galhos" para os nós filhos. As chaves são pedaços de palavras (as arestas compactadas).

3.1 Funcionamento da Inserção

A função `trie_insert` é recursiva. Para inserir uma palavra, ela segue uma lógica de três passos: primeiro, tenta encontrar um caminho existente na árvore para percorrer; se não encontra, tenta encontrar um nó para "dividir" e criar um novo galho; se nenhuma das opções anteriores for possível, ela cria um ramo totalmente novo para a palavra. Ao final do caminho, a frequência da palavra no documento é incrementada.

4 Módulos do Sistema

4.1 Indexação (`indexing.py`)

Este módulo prepara os dados e constrói o índice.

- **Pré-processamento:** A função `preprocess_text` padroniza o texto dos documentos, convertendo tudo para minúsculas e removendo pontuações com expressões regulares.
- **Construção do Índice:** A função `build_index_from_zip` lê o corpus, aplica o pré-processamento em cada texto e insere cada palavra na Trie.
- **Persistência do Índice:** Para que o servidor inicie rapidamente, criamos as funções `save_index_to_disk` e `load_index_from_disk`. Definimos um formato de texto próprio para salvar o índice: cada linha contém uma palavra, seguida de ponto e vírgula, e depois a lista de documentos e frequências (ex: `palavra;doc1:freq1,doc2:freq2`).

4.2 Recuperação da Informação (`search.py` e `relevance.py`)

Estes módulos são responsáveis por entender a busca do usuário e encontrar os melhores resultados.

4.2.1 Busca com Operadores Booleanos

A função `corpus_search` resolve as buscas com `AND`, `OR` e parênteses. Ela usa um algoritmo baseado no Shunting-yard, que utiliza pilhas para processar a consulta na ordem correta, respeitando a precedência dos operadores. As buscas por palavras na Trie retornam conjuntos de documentos, e os operadores `AND`/`OR` realizam operações de interseção e união nesses conjuntos.

4.2.2 Ranking com Z-Score

Depois de encontrar os documentos que correspondem à busca, a função `rank_by_relevance` os ordena. A ideia é que um documento é mais relevante se os termos da busca aparecem nele com uma frequência "anormalmente" alta.

- **Cálculo das Estatísticas:** Na inicialização do servidor, a função `calculate_corpus_stats` analisa a Trie inteira para calcular a frequência média e o desvio padrão de cada palavra em todo o corpus.
- **Cálculo do Z-Score:** Para cada documento, calculamos o z-score de cada termo da busca. A média desses z-scores se torna a "nota" de relevância daquele documento. Os resultados são então ordenados por essa nota.

4.3 Interface e Snippets (`main.py` e `utils.py`)

A interface foi feita com Flask.

- **Front-end:** As páginas de busca e resultados foram criadas com HTML e um CSS básico. Os resultados são paginados, mostrando 10 por página.
- **Snippet Inteligente:** Para tornar os resultados mais úteis, implementamos um snippet dinâmico. Para cada resultado, a função `find_best_term_for_snippet` decide qual palavra da busca é a mais importante (maior z-score) naquele documento. O snippet é então centralizado nessa palavra. Em seguida, a função `generate_snippet` destaca em negrito **todas** as palavras da busca que aparecem no trecho, dando mais contexto ao usuário.

5 Execução do Projeto

Para facilitar a execução da aplicação, criamos um `Makefile` simples. Os passos a seguir assumem que o avaliador possui **Python 3.9+**, **Git**, **make** e o framework **Flask** já instalados em seu sistema (ambiente Linux).

5.1 1. Obtenção do Código

Primeiramente, clone o repositório do projeto a partir do GitHub e navegue para a pasta criada:

```
$ git clone https://github.com/seu-usuario/seu-repositorio.git
$ cd nome-da-pasta-do-projeto
```

O arquivo `bbc-fulltext.zip`, que contém o corpus, já está incluído no repositório.

5.2 2. Execução da Aplicação com Makefile

Com o Flask já disponível no ambiente, a aplicação pode ser iniciada com um único comando no terminal:

```
$ make
```

Este comando executará a regra padrão do nosso `Makefile`, que invoca o servidor Flask em modo de depuração. O terminal exibirá as mensagens de inicialização do sistema.

Ao final do processo, a aplicação estará disponível para acesso no navegador, geralmente no endereço `http://12-7.0.0.1:5000`.

5.2.1 Funcionamento da Persistência do Índice

É importante notar que, na primeira vez que o comando `make` for executado, o sistema irá construir o índice a partir do zero e salvá-lo em disco como `my_corpus.idx`. Este processo inicial pode levar alguns segundos. Em todas as execuções subsequentes, o servidor carregará o índice diretamente deste arquivo, tornando a inicialização quase instantânea.

6 Dificuldades Encontradas

Alguns desafios surgiram durante o projeto:

1. **Implementação da Trie:** A lógica de inserção, especialmente a função para dividir um nó, foi o maior desafio algorítmico. A primeira versão tinha bugs e foi necessário reescrevê-la de forma recursiva para garantir que todos os casos fossem tratados corretamente.

2. **Gerenciamento de Dados no Flask:** No início, tentamos uma abordagem com upload de arquivos, mas percebemos que os dados (como a Trie) não eram mantidos entre as diferentes requisições do usuário. A solução foi adotar o modelo atual, que carrega e processa tudo na inicialização do servidor, o que se mostrou muito mais estável e eficiente.

3. **Snippet para Múltiplas Palavras:** Fazer o snippet funcionar bem para buscas com mais de um termo foi um desafio. A solução foi criar uma lógica que, para cada documento, escolhe o melhor termo para destacar, em vez de usar sempre o mesmo.

4. **Colaboração Através do Git:** Cada membro do grupo se encarregou de uma parte diferente do trabalho; Estas partes, no entanto, possuíam muitas interseções, o que gerou grandes desafios ao unir o código de cada um, atrasando o desenvolvimento do projeto.

5. **Restrições de Tempo e Escopo do Projeto:** O cronograma para o desenvolvimento foi desafiador, o que exigiu uma priorização rigorosa das funcionalidades essenciais em detrimento de refinamentos e recursos adicionais que gostaríamos de ter implementado. Embora tenhamos cumprido todos os requisitos obrigatórios, o tempo limitado restringiu nossa capacidade de explorar melhorias, como por exemplo, a qualidade da interface, já que queriamos ter aprimorado a experiência do usuário (UX) e o design visual.

7 Conclusão

Este trabalho nos permitiu aplicar na prática os conceitos estudados em sala de aula, resultando em um sistema de busca completo e funcional. Implementamos todos os requisitos do projeto, desde a estrutura de dados até a interface com o usuário.

Como melhorias futuras, poderíamos adicionar técnicas de Processamento de Linguagem Natural, como **Stemming**, para que a busca por "correr" também encontre documentos com "correu" ou "correndo". Isso tornaria a busca mais flexível e inteligente.

References

[1] Greene, D. and Cunningham, P. (2006). Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering. In *Proc. 23rd International Conference on Machine learning.*

[2] Pallets Projects. (2025). Flask Documentation. Disponível em: `https://flask.palletsprojects.com/`.

[3] Wikipedia. (2025). Shunting-yard algorithm. Disponível em: `https://en.wikipedia.org/wiki/Shunting_yard_algorithm`.