



**UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

ALEFI SANTOS CUNHA
LEAN HENRIQUE PEREIRA MIRANDA
LUCAS CASSIO COSTA
NICOLAS VON DOLINGER MOREIRA ROCHA

DOCUMENTAÇÃO MÁQUINA DE BUSCA

BELO HORIZONTE

2023

Visão Geral

Este documento apresenta a documentação do projeto da Máquina de Busca, um sistema de busca de documentos implementado pelo nosso grupo. Neste trabalho, buscamos desenvolver um sistema capaz de receber uma expressão de busca como entrada e retornar os documentos mais relevantes para a consulta fornecida. Para atingir esse objetivo, dividimos o sistema em três subsistemas principais: Coleta, Indexação e Recuperação.

Introdução

O subsistema de Coleta foi projetado considerando que os documentos de interesse já foram previamente coletados e estão armazenados em uma pasta específica. Portanto, a nossa responsabilidade principal foi a implementação dos subsistemas de Indexação e Recuperação.

Ao longo do desenvolvimento deste trabalho, o grupo enfrentou uma série de desafios técnicos e conceituais. Durante a implementação do subsistema de Indexação, tivemos que lidar com a criação de estruturas de dados eficientes para armazenar as informações relevantes dos documentos, como palavras-chave, índices e outras metainformações.

Já na etapa de implementação do subsistema de Recuperação, enfrentamos o desafio de projetar algoritmos de busca eficientes que pudessem processar as consultas do usuário e retornar os documentos mais relevantes de forma rápida e precisa.

Além dos desafios técnicos, o trabalho também envolveu uma forte colaboração em equipe. Foi necessário estabelecer uma comunicação efetiva, compartilhar ideias, dividir tarefas e realizar revisões periódicas para garantir que o projeto estivesse progredindo de acordo com os prazos estabelecidos. Acreditamos que o resultado final alcançado reflete nosso empenho e dedicação na busca por um sistema de busca eficiente e funcional.

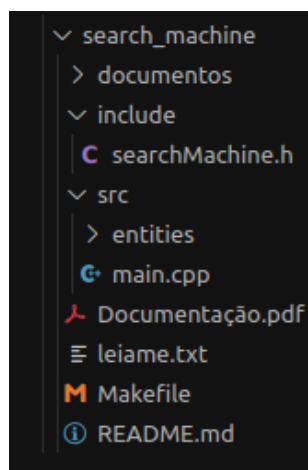
Implementação

Para garantir uma organização eficiente e facilitar o trabalho do grupo, o

projeto foi estruturado em diversas pastas dentro do diretório principal chamado "search_machine". Essas pastas são: "build", "src", "include" e "documentos". Vamos detalhar a função de cada uma delas:

- build: Esta pasta é responsável por armazenar os executáveis do projeto após a compilação. Ela é criada somente após a compilação ser concluída e serve para guardar os arquivos executáveis resultantes.
- include: Nesta pasta, é possível encontrar o contrato de implementação da classe "Search Machine". Aqui estão armazenados os arquivos de cabeçalho (header files) que descrevem a interface pública da classe, ou seja, as declarações dos métodos, atributos e estruturas de dados utilizados pela classe.
- src: Esta pasta contém a implementação da classe "Search Machine" e o arquivo "main.cpp". Além disso, dentro da pasta "src", você pode encontrar a pasta "entities", que armazena a implementação específica dos métodos da classe "Search Machine". É nessa pasta que membros do grupo trabalharam no desenvolvimento dos recursos e funcionalidades da aplicação.
- documentos: A pasta "documentos" é destinada ao armazenamento dos documentos fornecidos pelo usuário para realizar as buscas desejadas.

Essa estrutura de pastas ajuda a organizar o projeto de forma clara e coerente, permitindo que qualquer um encontre facilmente os arquivos e recursos presentes no projeto.



Para resolver o problema de implementação da Máquina de Busca, utilizamos a linguagem de programação C++ e desenvolvemos uma classe chamada SearchMachine. Nesta seção, iremos explicar de forma clara e objetiva como o problema foi resolvido, justificando os algoritmos e as classes utilizadas.

Segue abaixo como foi implementado o Search Machine.h:

```
1  #ifndef searchMachine
2  #define searchMachine
3
4  #include <iostream>
5  #include <vector>
6  #include <string>
7  #include <map>
8
9  using std::string;
10 using std::map;
11 using std::vector;
12 using std::pair;
13
14 class SearchMachine {
15 public:
16     // Construtor da Classe;
17     SearchMachine();
18
19     // Lê os arquivos
20     void readFile();
21
22     // Subsistema de indexação
23     map<string, map<string, int>> buildIndex(string newWord, string arquivo);
24
25     // Subsistema de recuperação
26     vector<pair<string, int>> search(string words);
27
28 private:
29     // Normaliza as palavras (coloca por padrao todas minúsculas)
30     string normalizeWord(string word);
31
32     // Indice invertido
33     map<string, map<string, int>> invertedIndex_;
34
35     // caminho de /documentos
36     string documentsPath_;
37 };
38
39 #endif
```

A classe SearchMachine possui os seguintes atributos e métodos:

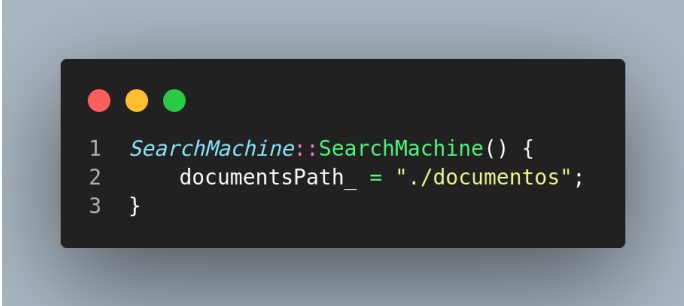
Atributos

invertedIndex_: Um mapa que representa o índice invertido, responsável por armazenar as palavras-chave encontradas nos documentos e seus respectivos arquivos e contagens de ocorrências.

documentsPath_: Uma string que representa o caminho da pasta onde os documentos estão armazenados.


Métodos

SearchMachine(): O construtor da classe, responsável por inicializar o atributo “documentsPath_” com o caminho padrão “./documentos”.



```
1 SearchMachine::SearchMachine() {
2     documentsPath_ = "./documentos";
3 }
```

normalizeWord(): Um método privado que recebe uma palavra como entrada e a normaliza, convertendo todas as letras para minúsculas e removendo caracteres especiais. Isso é feito para garantir uma correspondência correta durante a indexação e recuperação dos documentos.



```
1 string SearchMachine::normalizeWord(string word) {
2     std::string normalized = "";
3
4     for (char c : word) {
5         if (isalpha(c)) {
6             normalized += tolower(c);
7         }
8     }
9
10    return normalized;
11 }
```

buildIndex(): Um método que recebe uma nova palavra-chave e o nome do arquivo onde ela foi encontrada. Esse método atualiza o índice invertido (`invertedIndex_`), incrementando a contagem de ocorrências dessa palavra-chave no documento correspondente.

```
1 map<string, map<string, int>> SearchMachine::buildIndex(string newWord, string arquivo){
2     invertedIndex_[newWord][arquivo]++;
3     return invertedIndex_;
4 }
```

readFile(): Um método que percorre os arquivos presentes na pasta de documentos especificada pelo atributo `documentsPath_`. Para cada arquivo encontrado, o método lê cada linha e extrai as palavras presentes. Em seguida, normaliza essas palavras e utiliza o método `buildIndex()` para atualizar o índice invertido com as informações relevantes.

```
1 void SearchMachine::readFile() {
2     for (const auto arquivo : filesystem::directory_iterator(documentsPath_)) {
3         if (arquivo.is_regular_file()) {
4             ifstream arquivoEntrada(arquivo.path());
5             if (arquivoEntrada) {
6                 string linha;
7                 while (getline(arquivoEntrada, linha)) {
8                     istringstream iss(linha);
9                     string palavra;
10                    while (iss >> palavra) {
11                        string fileName = arquivo.path().stem();
12                        string newWord = normalizeWord(palavra);
13                        invertedIndex_ = buildIndex(newWord, fileName);
14                    }
15                }
16                arquivoEntrada.close();
17            }
18        }
19    }
20 }
21
```

search(): O subsistema de recuperação, que recebe uma string de consulta como entrada. Primeiro, normaliza as palavras da consulta e armazena-as em um vetor. Em seguida, percorre cada palavra e consulta o índice invertido (`invertedIndex_`) para obter os documentos que contêm essa palavra. O método mantém um mapa de ocorrências de documentos relevantes, contando quantas palavras-chave foram encontradas em cada documento. Em seguida, filtra os documentos que não possuem todas as palavras da consulta e ordena os documentos restantes com base no número de ocorrências e em ordem lexicográfica, retornando um vetor de pares (nome do arquivo, número de ocorrências) ordenado.

```
1 vector<pair<string, int>> SearchMachine::search(string input) {
2     // Normaliza as palavras da consulta
3     istream iss(input);
4     vector<string> words;
5     string word;
6     while (iss >> word) {
7         string normalizedWord = normalizeWord(word);
8         words.push_back(normalizedWord);
9     }
10
11     // Mapa para armazenar o número de ocorrências de cada documento relevante
12     map<string, int> documentOccurrences;
13     map<string, int> wordOccurrences;
14     map<string, int> totalOccurrences;
15     // Percorre cada palavra da consulta
16     for (auto it = words.begin(); it != words.end(); it++) {
17         // Consulta o índice invertido para obter os documentos que contêm a palavra
18         if (invertedIndex_.find(*it) != invertedIndex_.end()) {
19             wordOccurrences = invertedIndex_[*it];
20             // Atualiza o contador de ocorrências de cada documento
21             for (auto entry = wordOccurrences.begin(); entry != wordOccurrences.end(); ++entry) {
22                 documentOccurrences[entry->first]++;
23                 totalOccurrences[entry->first] += entry->second;
24             }
25         } else {
26             // Se alguma palavra não for encontrada no índice, retorna um vetor vazio
27             return vector<pair<string, int>>();
28         }
29     }
30
31     for(auto entry = documentOccurrences.begin(); entry != documentOccurrences.end(); ++entry) {
32         if(entry->second != words.size()) {
33             totalOccurrences.erase(entry->first);
34         }
35     }
36
37     vector<pair<string, int>> sortedOccurrences(totalOccurrences.begin(), totalOccurrences.end());
38     sort(sortedOccurrences.begin(), sortedOccurrences.end(), [](const pair<string, int>& a, const pair<string, int>& b) {
39         if (a.second != b.second) {
40             return a.second > b.second; // Sort by number of occurrences (descending order)
41         } else {
42             return a.first < b.first; // Sort by lexicographic order (ascending order)
43         }
44     });
45
46     return sortedOccurrences;
47 }
48
49 }
```

Justificativas

- Utilizamos um mapa (`invertedIndex_`) para representar o índice invertido, pois essa estrutura de dados nos permite armazenar as palavras-chave e seus arquivos correspondentes de forma eficiente. Além disso, o mapa facilita a recuperação das informações durante o processo de busca.
- O método `normalizeWord()` foi implementado para garantir que as palavras-chave sejam tratadas de forma consistente, independentemente de diferenças em maiúsculas/minúsculas ou caracteres especiais. Isso evita inconsistências durante a indexação e recuperação dos documentos.
- Durante a indexação dos documentos, o método `buildIndex()` atualiza o índice invertido com as informações relevantes. Utilizamos a estrutura de dados `map` aninhada para armazenar as palavras-chave, os arquivos e as contagens de ocorrências, permitindo um acesso rápido e eficiente aos dados.
- No subsistema de recuperação, o método `search()` utiliza uma abordagem de consulta por palavras-chave, verificando a presença de cada palavra na consulta no índice invertido. Essa abordagem nos permite identificar os documentos mais relevantes com base na contagem de ocorrências de cada palavra-chave.

Em resumo, a implementação da Máquina de Busca utilizando a classe `SearchMachine` em C++ demonstra um processo eficiente de coleta, indexação e recuperação de documentos. A escolha adequada das estruturas de dados e algoritmos utilizados permite a busca rápida e precisa dos documentos relevantes para uma determinada consulta.

Conclusão

Ao finalizar a implementação da Máquina de Busca, podemos concluir que o trabalho foi bem-sucedido em alcançar o objetivo proposto: desenvolver um sistema de busca de documentos eficiente e capaz de retornar resultados relevantes para consultas fornecidas.

Durante o processo de desenvolvimento, enfrentamos diversos desafios e superamos obstáculos técnicos e conceituais. A implementação dos subsistemas de Indexação e Recuperação exigiu um estudo aprofundado das melhores práticas e algoritmos disponíveis, além da criação de estruturas de dados otimizadas para armazenar e processar as informações dos documentos.

A linguagem de programação C++ se mostrou uma escolha adequada para o desenvolvimento do projeto, proporcionando um bom desempenho e flexibilidade na manipulação dos dados. Através da aplicação dos conhecimentos adquiridos em programação, estruturas de dados e algoritmos de busca, conseguimos construir um sistema robusto e eficiente.

Além disso, a colaboração em equipe foi essencial para o sucesso do projeto. O trabalho conjunto, a troca de ideias e a divisão de tarefas permitiram um avanço consistente e organizado, garantindo o cumprimento dos prazos estabelecidos e a qualidade do sistema desenvolvido.

No entanto, reconhecemos que sempre há espaço para melhorias e otimizações. Em projetos futuros, poderíamos explorar técnicas avançadas de processamento de linguagem natural, como stemming, lematização e análise semântica, para aprimorar ainda mais a relevância dos resultados retornados pelo sistema de busca.

Em suma, a experiência adquirida com a implementação da Máquina de Busca foi extremamente valiosa para o nosso grupo. Através deste projeto, expandimos nossos conhecimentos e habilidades em programação, resolução de problemas complexos e trabalho em equipe.