Documentação

1. Introdução

Para este trabalho, foi proposto a implementação de dois dos três componentes existentes em uma máquina de busca, o indexador e o processador de consultas.

Será passado como parâmetro para o programa o nome do arquivo contendo a consulta , nome do arquivo de saída, corpus (conjunto de documentos) e um arquivo contendo as stopwords.

Com esses parâmetros em mãos deveremos montar o indexador invertido, tratando as stopwords contidas em cada documento e a estrutura de dados utilizados em sua armazenagem.

No processador de consulta temos como objetivo final retornar os documentos mais relevantes, precisaremos computar uma tabela de frequência, com as frequências que cada palavra aparece no documento.

2.Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Ambiente de Desenvolvimento

Sistema Operacional: Pop!_OS 21.10

Compilador: g++ versão 11.2.0

Processador: AMD Ryzen 5500u

Memória RAM: 8GB

2.2 Estrutura de Dados

2.2.1 Classes

Para a armazenagem de dados foram utilizadas diversas estruturas de dados. As principais estruturas de dados utilizadas foram lista encadeada, lista encadeada que só aceita termos únicos e o hash.

2.2.2 Classe Lista Encadeada

Lista Encadeada foi utilizada na armazenagem do vocabulário, das palavras de consulta, das stopword e indexes que faz o link entre as consulta e a matriz de peso, motivo de se utilizar essa estrutura de dados é devido a impossibilidade de se saber o tamanho de antemão.

Ela é utilizada como base para a classe Lista Encadeada Única e para o armazenamento das stopwords.

2.2.3 Classe Lista Encadeada Única

Esta classe é herdada da Lista Encadeada, porém a inserção verifica se o elemento está presente na lista, caso ele não esteja presente o elemento é inserido, caso contrário ele não é inserido.

Ela é utilizada na armazenagem das palavras do vocabulário, e consulta. Pois a mesma palavra é inserida múltiplas, não é necessário que seja armazenado repetição de palavras.

2.2.4 Classe Hash

Para o armazenamento dos índices inversos foi utilizado a estrutura de dados hash com lista encadeada. A estratégia para lidar com colisões foi resolução de colisão por encadeamento aberto, pois o tamanho do hash é 12.400.001 e portanto a grande maioria dos dos hashs não serão utilizadas. Em caso de colisão ele irá procurar a próxima posição vazia.

Foi utilizado para a armazenagem do índice inverso, devido a auto eficiência na procura de uma palavra que tenha sido incluída na tabela hash.

A escolha do hash foi o somatório do valor de cada carácter no na tabela ASCII vezes a sua posição na string. Para se evitar o overflow somente sete primeiros caracteres da palavra foram considerados.

2.3 Matriz esparsa

Essa estrutura de dados não foi implementada em classe, mas está presente na função main do programa.

Ela foi utilizada devido a grande quantidade de elementos nulos presentes na tabela de pesos (matriz de pesos)

Esta implementação consiste em uma matriz com três linhas. A primeira armazena qual seria o índice i e a segunda guada índice j e a terceira linha o valor, esses índices dizem em qual posição estaria o elemento em uma matriz normal. O n

3 Análise de Complexidade

Para a analise de complexidade de espaço, variáveis criadas para a armazenagem de strings serão consideradas O(1)

3.1 Main

void insert_word() → Insere as palavras tratadas na lista de vocabulários

Complexidade de Tempo: O(mn)

- Pois, as instruções são O(1), dado que não percorrem nenhum loop e o método vocabulary.insert() é O(mn), com m sendo o numero de palavras contidas no vocabulário e n sendo a quantidade de letras que a palavra tem.

Complexidade de Espaço: O(1)

- Pois, é criada somente uma variável.

int get_number_of_files() → Conta quantos documentos existem no corpus

Complexidade de Tempo: O(n)

- Pois, itera sobre cada documento.

Complexidade de Espaço: O(1)

- pois é criado somente uma variável.

void open_corpus() → Abre o documento trata as palavras e as inseri em um novo arquivo

Complexidade de Tempo: $O(n^4)$

- Possui três loops, o loop externo itera sobre a cada documento é chamado a função get_document_index e suas instruções são O(1), e portanto seu bloco de comando excluindo os loops internos é O(1).
- O segundo loop itera sobre cada palavra contida no documento, neste loop são chamadas as funções to_lowercase O(n) (n em função das letras da palavra) , is_stopword O(n) (n em função do número de stopwords) , e o a função insert_word O(n) (n em função do tamanho do vocabulário).
- E por final o ultimo loop, que itera sobre cada carácter da palavra chamando a função insert_word O(n) e alguns métodos da classe string que serão considerados O(1).

Complexidade de Espaço: O(n)

- Pois, é criado um vetor, cujo o tamanho é o tamanho da palavra

int get_document_index() → Pega a parte numérica do nome do arquivo

Complexidade de Tempo: O(1)

- O método find da classe string será considerado como O(1), portando caso o nome do arquivo tenha “.txt”, isso será removido.

Complexidade de Espaço: O(1)

• É criado um numero fixo de variáveis.

void read_stopwords() → Insere as palavras do arquivo de consulta na lista de stopwords.

Complexidade de Tempo: O(n)

• Itera sobre cada palavra do arquivo stopword, inserindo a palavra na lista de stopword, cuja a complexidade é O(1).

Complexidade de Espaço: O(1)

• Pois no escopo da função é criado um numero fixo de variáveis.

void read_search_file() → Insere as palavras do arquivo de consulta na lista de palavras de consulta

Complexidade de Tempo e Espaço é análogo a função acima.

void to_lowercase()→ Reescreve a palavra para caixa baixa

Complexidade de Tempo: O(n)

• Pois, itera sobre cada letra, convertendo-a para caixa baixa.

Complexidade de Espaço: O(1)

• Pois não é criado nenhuma nova variável.

bool is_stopword() → Verifica se uma palavra é uma stopword

Complexidade de Tempo: O(n)

• Compara a palavra com cada stopword da lista de stopword. Comparação da string será considerado como O(1).

Complexidade de Espaço: O(1)

• Pois, é criado um numero fixo de variáveis.

void remove_special_characters(s) → Marcar quais caracteres deverão ser mantidos

Complexidade de Tempo: O(n)

• Itera sobre cada carácter, caso ele esteja entre a e z, ele será um carácter valido.

Complexidade de Espaço: O(1)

• Pois, é criado um numero fixo de variáveis.

void insert_word() → Insere a palavra no vocabulário que é uma lista de palavras.

Complexidade de Tempo: O(n)

- Pois, apesar de não possuir nenhum loop a função chama o método insert da classe lista única, cuja complexidade é O(n).

Complexidade de Espaço: O(1)

- Pois no escopo de função é criado um numero fixo de variáveis.

void inverse_index_gen() → Cria o índice inverso

Complexidade de Tempo: $O(n^2)$

- Realiza um loop sobre cada documento, chamando a os métodos get_hash O(1) da classe hash e o método increment O(n) da classe lista única, com n sendo o numero de pares (documento, frequência).

Complexidade de Espaço: O(1)

- Pois no escopo da função é criado um numero fixo de variáveis.

template <typename T>

T create_matrix()**→ Cria uma matriz de alocação dinâmica.

Complexidade de Tempo: O(n)

- Pois realiza um loop sobre cada linha da matrix;

Complexidade de Espaço: $O(n^2)$

- Pois, a memória é aloca para uma matrix;

template <typename T>

void initialize_matrix() → Inicializa os elementos da matriz com 0.

Complexidade de Tempo: O(n^2)

- Pois, é necessário iterar sobre o numero de linhas e numero de colunas, inicializando o valor da célula (elemento) da matriz como 0.

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável.

template <typename T>

void delete_matrix() → Desaloca a memória utilizada na matriz.

Complexidade de Tempo: O(n)

- Itera sobre as linhas, desalocando a memória utilizada.

Complexidade de Espaço: O(1)

- Pois, não é criado nenhuma nova variável.

unsigned int number_of_elements_that_is_ non_zero () → Conta o numero de pares (documento, id) que a tabela hash possui.

Complexidade de Tempo: O(n)

- Itera sobre o vocabulário, para cada palavra é chamado o método get_hash O(1) e get_hash_size O(1).

Complexidade de Espaço: O(1)

- Pois, é criado um numero fixo de variáveis

void document_weight_gen() → Calcula o peso de cada de cada par (documento, frequência).

Complexidade de Tempo: $O(n^2)$

- Itera sobre o vocabulário, para cada palavra é chamado os métodos get_hash O(1), get_first_element O(1) e get_hash_size O(1), no loop interno é iterado sobre o numero de arquivos, calculando o peso do par (documento, frequência) O(1).

Complexidade de Espaço: O(1)

- É criado um numero fixo de variáveis.

void search_weight_gen() → Gera a matriz binária da consulta

Complexidade de Tempo: $O(n^2)$

- Possui dois loops aninhados, o loop externo itera sobre o lista de palavras de consulta, para cada palavra é obtido o seu hash através do método get_hash O(1), no loop interno é iterado sobre cada documento cujo o bloco de comando é O(1).

Complexidade de Espaço: O(1)

- Pois, é criado um numero fixo de variáveis.

void search_sum_weights() → Calcula o peso da consulta não normalizada.

Complexidade de Tempo: O(n) → caso médio

- No pior caso será $O(n^2)$, mas isso é praticamente impossível, a não ser que todos os documento tenham as mesma palavras e a consulta seja o

vocabulário. Neste caso iteraria sobre o vocabulário, o numero de documentos e o bloco de comandos é O(1)

- Considerando o caso médio que o numero de tuplas (palavra, documento, frequência) seja a raiz de vocabulário x documento. Teremos que está função tem custo O(n). Pois apesar de possuir vários loops, indiretamente eles estão em função de somente um variável que é o numero de tuplas (palavra, documento, frequência).

Complexidade de Espaço: O(1)

- É criado um numero fixo de variáveis

void search_sum_weights_norm() → Normaliza o vetor com os pesos da consulta

Complexidade de Tempo: O(n)

• Possui três loops não aninhados, seus blocos de comando s são O(1), porem o primeiro loop é iterados mais vezes.

Complexidade de Espaço: O(n)

• É criado um vetor que depende do tamanho do corpus.

void find_indexes() → Encontra os índices das palavras na matriz peso

Complexidade de Tempo: $O(n^2)$

• Itera sobre a lista de palavra de consulta, chamando a função find_index() $O(n^2)$. No pior caso quando lista com as palavra de consulta for grande $O(n^3)$. Caso médio $O(n^2)$, pois a consulta tende a ter um numero pequeno de palavras.

Complexidade de Espaço: O(1)

• É criado um numero fixo de variáveis.

void find_index() → Encontra em que posição a palavra está na matriz de peso.

Complexidade de Tempo: $O(n^2)$

• Itera sobre vocabulário, para cada palavra do vocabulário que for igual ao parâmetro word, será chamado o método insert da classe lista única O(n).

Complexidade de Espaço: O(1)

• Pois no escopo da função é criado um numero fixo de variáveis.

void list_to_array() → Converte uma lista encadeada para uma lista estática

Complexidade de Tempo: O(n)

• Itera sobre a lista estática, cada elemento é colocado em uma posição da lista estática, com custo O(1).

Complexidade de Espaço: O(1)

• Pois, é criado um numero fixo de variáveis.

void greater_relevance() → Ordena e imprimi os 10 primeiros documentos mais relevantes.

Complexidade de Tempo: $O(n^2)$

• Chama o função inserção $O(n^2)$ que ordena o vetor, imprimi os 10 primeiros documentos mais relevantes.

Complexidade de Espaço: O(1)

• É criado um numero fixo de variáveis.

3.2 Hash

3.2.1 Hash → Classe Pai

Hash_String() → Construtor da Classe hash

Complexidade de Tempo: O(1)

• Pois inicializa as variáveis, sem utilização de loops.

Complexidade de Espaço: O(n)

• Cria vetores que dependem do valor de n.

~Hash_String() → Destrutor da Classe hash

Complexidade de Tempo: O(1)

• Realiza desalocação de memoria utilizando delete[], sem utilização de loops;

Complexidade de Espaço: O(1)

• Não é criado nenhuma variável nova.

ul_int hash_it() → Calcula qual seria o hash da palavra

Complexidade de Tempo: O(n)

• Pois o for é executado n vezes, com n sendo o tamanho da string e o bloco de comando do for é O(1)

Complexidade de Espaço: O(1)

• Pois são criadas somente duas novas variáveis , que não dependem do valor de n, com n sendo o tamanho da string.

virtual ul_int get_hash() → função deverá ser implementado quando for herdada

Complexidade de Tempo: O(1)

• Pois imprime uma mensagem no terminal

Complexidade de Espaço: O(1)

• Pois não é criado nenhuma nova variável

void remove() → mesmo caso da função acima

Complexidade de Tempo: O(1)

Complexidade de Espaço: O(1)

Hash_String_Pair() → Construtor da classe hash

Complexidade de Tempo: O(1)

• Pois chama o construtor da classe pai O(1), e faz alocação de memória com o operador new.

Complexidade de Espaço: O(n)

• Pois, faz aloca memória em função de n.

~Hash_String_Pair() → Destrutor da Classe herdada

Complexidade de Tempo: O(1)

• Pois, o destrutor da classe pai é O(1) e realiza desalocação de memoria utilizando delete[], sem utilização de loops;

Complexidade de Espaço: O(1)

• Pois, o método só desaloca memória não é criado nenhuma nova variável, em ambos os métodos destrutores.

ul_int get_hash() → retorna em qual hash a palavra está armazenada

Complexidade de Tempo: O(1)

• Pois, como a tabela hash.

Complexidade de Espaço: O(1)

• Pois, é criado somente uma nova variável.

Cell<Pair>* get_first_element() → Retorna o ponteiro do primeiro elemento da lista que está contido neste hash.

Complexidade de Tempo: O(1)

• Pois, este método só chama o método get_primeiro_elemento da classe Lista, cuja a complexidade é O(1).

Complexidade de Espaço: O(1)

• Pois, este método não cria nenhuma nova variável, somente retorna um ponteiro.

int get_hash_size() → Retorna o tamanho da lista que está contida dentro desse hash.

Complexidade de Tempo: O(1)

• Pois, este método só chama o método get_tamanho da classe Lista, cuja a complexidade é O(1).

Complexidade de Espaço: O(1)

• Pois, este método não cria nenhuma nova variável, somente retorna um inteiro.

void initialize(Lista<Word> &list) → Inicializa a tabela hash com as palavras do vocabulário

Complexidade de Tempo: O(n)

• Pois, devido as características da tabela hash o loop interno é O(1), dado que rapidamente ele achará uma posição vazia. O loop externo executará em função do tamanho do vocabulário e seu bloco de comando é O(1).

Complexidade de Espaço: O(1)

• Pois, no escopo da função não é criado nenhuma variável que dependa de algum valor.

void insert() → Insere um elemento na lista de elementos desse hash

Complexidade de Tempo: O(1)

• Pois, o método chama o método insert da Classe Lista_única, cuja a complexidade é O(1) e o resto do bloco de comandos só atribui novos valores à variáveis, sem a utilização de um loop.

Complexidade de Espaço: O(1)

- Pois, nenhuma variável foi criada.

void increment() → Incrementa a frequência do Par (palavra, documento) por 1.

Complexidade de Tempo: O(n)

- Pois, o loop é executado em função do tamanho da lista que está contido dentro do hash, que no limite pode ser igual ao tamanho do vocabulário e seu conjunto de comandos é O(1). Caso não encontre, será inserido o documento na lista, utilizando o método insert da própria classe hash com custo O(1).

Complexidade de Espaço: O(1)

- Pois, é criado somente uma nova variável

3.3 Lista

Lista() → Construtor da Classe Lista

Complexidade de Tempo: O(1)

- Executa um numero fixo de vezes, não percorre nenhum loop

Complexidade de Espaço: O(1)

- Cria um numero fixo de variáveis.

~Lista() → Destrutor da classe lista

Complexidade de Tempo: O(n)

- Chama o método remove tudo, cuja complexidade é O(n).

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável

int get_tamanho() → retorna o valor do atributo tamanho

Complexidade de Tempo: O(1)

- Retorna o valor do atributo tamanho.

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável

Cell<U>* get_primeiro_elemento() → Retorna um ponteiro para o primeiro elemento da lista

Complexidade de Tempo: O(1)

- Executa um número fixo de vezes.

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável.

Cell<U>* get_ultimo_elemento() → Retorna um ponteiro para o último elemento da lista.

Complexidade de Tempo: O(1)

- Retorna o atributo fim.

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável.

void inseri_no_inicio() → Insere um elemento no inicio da lista

Complexidade de Tempo: O(1)

- É executado um numero fixo de vezes, pois é criado uma nova célula que apontará para o primeiro elemento da lista e o inicio da lista apontara para a célula.

Complexidade de Espaço: O(1)

- É criado um numero fixo de variáveis, cujo o tamanho é fixo.

void inseri_no_fim() → Insere um elemento no final da lista

Complexidade de Tempo: O(1)

- Pois, é criado uma nova célula e o ultimo elemento apontará para essa nova célula.

Complexidade de Espaço: O(1)

- É criado um numero fixo de variáveis, cujo o tamanho é fixo.

U remove_no_inicio() → Remove o primeiro elemento da lista.

Complexidade de Tempo: O(1)

- É criado uma nova variável que irá armazenar o primeiro elemento, o inicio irá apontar para o segundo elemento e por fim é apagado o primeiro elemento.

Complexidade de Espaço: O(1)

- É criado um numero fixo de variáveis, cujo o tamanho é fixo.

void remove_tudo() → Remove todos os elementos da lista

Complexidade de Tempo: O(n)

- Executa o método remove no inicio em função do tamanho da lista.

Complexidade de Espaço: O(1)

- Não é criado nenhuma nova variável.

3.4 Lista Única

Unic_List() → Construtor da classe lista única.

Complexidade de Tempo: O(1)

- Chama o construtor da classe Lista, cuja complexidade é O(1)

Complexidade de Espaço: O(1)

- Não é criado nenhuma variável

~Unic_List() → Destrutor da classe lista única

Complexidade de Tempo: O(n)

- Chama o destrutor da classe lista, cuja complexidade é O(n).

Complexidade de Espaço: O(1)

- Não é criado nenhuma variável.

bool insert() → Insere o elemento, caso ele ainda não esteja presente na lista

Complexidade de Tempo: O(n)

- Pois, a lista é percorrida linearmente a verificando se o elemento já inserido caso ainda não é chamado o método insere no fim, cuja complexidade é O(1).

Complexidade de Espaço: O(1)

- Pois no escopo do método só é criado um número fixo de variáveis.

4 Estratégia de Robustez

Foi utilizado neste projeto a biblioteca msgassert para verificar inconsistências e pré-condições.

É verificado em todos os arquivos abertos se foram abertos corretamente, caso contrário o programa é interrompido e uma mensagem será imprimida no terminal.

Função parse_args:

Verifica se os parâmetro passados para o programa estão corretos, caso contrário o programa será interrompido.

Funções que utilizam o método get_hash:

É verificado se o valor retornado é maior que 0, pois como o objeto da classe Hash, já foi inicializado get_hash e o hash 0 não é utilizado, o método só deverá retornar valores maiores que 0.

Somente as principais funções

Função search_weight_gen():

Verifica no loop interno se o índice j é menor que o documento contido no par (documento, frequência), caso contrário o programa é interrompido e uma mensagem é exibida no terminal, pois isso significa que que os pares (documento, frequência), não foram inseridas em ordem.

Função list_to_array():

Verifica se o numero de interações é igual ao tamanho da lista, caso contrário o a função é interrompida.

5 Análise Experimental

6 Conclusão

Este programa utilizou vários conceitos estudos durante o curso de Estruturas de Dados, como lista encadeada, algoritmo de ordenação e tabela hash.

Um dos maiores desafios ao se implementar o programa foi o tempo, apesar da haver duas semanas. Outros problemas enfrentados foram a implementação do hash, tratamento do das palavras devido as diversas possíveis interpretações, nomes dos documentos não serem sequencias e por último a implementação da matriz esparsa.

Existem diversas melhorias a serem feitas no código, o algoritmo de ordenação escolhido foi o inserção devido a sua fácil implementação, porém para uma maior eficiência poderia utilizar o heapsort, para encontrar os 10 maiores. Reestruturação das funções para uma maior legibilidade.

7 Instruções de Compilação e Execução

Compilação:

No Linux, abra, pelo terminal, o diretório que contém o arquivo Makefile e digite o comando

make. Tal comando irá criar o programa juntamente com arquivos necessários para a

compilação.

Execução:

Para a execução do código digite no terminal.

<diretório do programa>./bin/programa -i <par_1> -o <par_2> -c <par_3> -s <par_4>

<par_1> → deve ser o nome do arquivo de entrada contendo as palavras de consulta, parâmetro obrigatório.

<par_2> → deve ser o nome do arquivo de saída, parâmetro obrigatório.

<par_3> → deve ser o caminho para o corpus, parâmetro obrigatório.

<par_4> → deve ser o numero do arquivo contendo as stopwords, parâmetro obrigatório.

Os parâmetros devem ser inseridos nesta em ordem, para que o programa funcione corretamente.

Durante a execução será criado vários arquivos na pasta do programa com as palavras tratadas, caso queira que os arquivos sejam criados em outro diretório.

```
#define PROCESS_FILE_FOLDER  "" // exemplo Processo/  -> dont forget the / at the end
```

Insira o caminho da pasta no arquivo main.cpp.

Referências

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade

Federal de Minas Gerais. Belo Horizonte.