UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

LUCAS FERREIRA DOS SANTOS

RIAN WAGNER COSTA

ALGORITMO PARA IDENTIFICAÇÃO DE REPETIÇÃO E FREQUÊNCIA DE PALAVRAS UTILIZANDO HASH TABLE.

Documentação completa

SÃO JOÃO DEL REI - MG

DEZEMBRO, 2022

SUMÁRIO

1 INTRODUÇÃO

A popularização da web e o grande número de compartilhamentos de documentos fez com que o processamento de textos fosse inerente à evolução da tecnologia [1]. A quantidade de dados por textos aumentou drasticamente e a tendência é que continue crescendo aceleradamente.

Por isso, saber processar longas cadeias de caracteres de maneira performática é extremamente importante e aplicável em diversas áreas, pois tal aplicação acaba por ser tarefa base para diversas aplicações: [2]:

- Detecção de plágio: Considerando as cadeias de caracteres do alfabeto é possível realizar combinações que podem resultar variações textuais presentes em dois arquivos textuais diferentes. É possível detectar o plágio por meio da similaridade das estruturas e sequências de caracteres.

- Algoritmos de combinação de bases moleculares: Através da sinalização de bases moleculares, o processamento dos dados de entrada fornece uma sequência como o DNA, por exemplo.

Fazendo uso dessas aplicabilidades, o objetivo deste trabalho é detectar palavras repetidas em um texto e, caso positivo, sinalizar a localização e quantas vezes essa cadeia de caractere foi repetida.

2 DESENVOLVIMENTO

O problema proposto considera um algoritmo de leitura de dados que processe um texto com “*n”* linhas e “*x”* palavras. Partindo de um outro arquivo com palavras a serem pesquisadas, cabe ao algoritmo sinalizar palavras repetidas, onde em quais linhas elas estão localizadas e quantas vezes são repetidas.

Alguns requisitos foram sinalizados para a construção da aplicação:

- As palavras formadas por uma única letra devem ser ignoradas, bem como espaços em branco, sinais de pontuação e caracteres especiais.

- As palavras apresentadas não terão acentuação.

- O programa deve se *“case insensitive"* ou seja, não deve diferenciar letras maiúsculas de minúsculas.

- A estrutura para pesquisa e inserção deverá ser uma “*HASH TABLE* “ e colisões devem ser tratadas por endereçamento aberto.

2.1 ESTRUTURA DA HASH

O hash fornece pesquisa em tempo constante, operações de inserção e exclusão em média. É por isso que o hashing é uma das estruturas de dados mais usadas, problemas de exemplo são, elementos distintos , contando frequências de itens, encontrando duplicatas [3].

Na aplicação em questão, a hash armazenará palavras do arquivo de texto bem como, as linhas em que ela se repete. Para isso, foi necessário a criação de uma TAD para obter essas informações, como pode ser visto na figura 1.

Figura 1 - TAD de registro de dados

```
3   typedef struct {
4       char chave[24];   // A palavra em letras minusculas
5       int  ocorrencias; // Quantas vezes a palavra aparece no texto
6       int  linhas[25];  // Em quais linhas a palavra aparece
7   } Registro;
```

Fonte: os autores

Para armazenar todas os dados na hash, foi necessário a criação de um registro auxiliar com a estrutura de vetor, veja na figura 2:

Figura 2 - Registro auxiliar

```
9   typedef struct {
10      Registro registros[256];
11  } Tabela;
```

Fonte: os autores

Como função de hashing, utilizamos a “DJB2” por se mostrar mais eficiente e simples, veja na figura 3:

Figura 03 - Função de Hashing “DJB2”

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

Fonte: http://www.cse.yorku.ca/~oz/hash.html

2.2 FUNÇÕES

2.2.1 FUNÇÃO TRATAMENTO DE COLISÕES

Tratando-se de hash, foi necessário lidarmos com as colisões de posições. A função *Tabela_Busca* percorre as posições da hash e caso esteja vazia ou com a chave correspondente, veja na figura 04:

Figura 04 - Função *Tabela_Busca*

```c
Registro *Tabela_buscar(Tabela *tabela, const char *chave) {
    int indice = hashCode(chave) % 256;
    // Percorre a tabela até encontrar um registro vazio ou com a mesma chave
    while (strcmp(tabela->registros[indice].chave, "") != 0 && strcmp(tabela->registros[indice].chave, chave) != 0) {
        indice = (indice + 1) % 256;
    }
    // Se o registro estiver vazio, coloca a chave nele
    if (strcmp(tabela->registros[indice].chave, "") == 0) {
        strcpy(tabela->registros[indice].chave, chave);
    }
    return &tabela->registros[indice];
}
```

Fonte: os autores

2.2.2 FUNÇÃO PARA ANÁLISE DE ENTRADA

A função *analisar_entrada* lê o arquivo *input.txt,* definido como forma padrão, veja na figura 5. Através de estruturas condicionais e de repetição, são definidos os pontos de parada para análise das palavras, veja na figura 6:

Figura 5 - Função *analisar_entrada*

```c
void analisar_entrada(Tabela *tabela, FILE *entrada) {
    int  numLinha = 1; // Número da linha atual
    char linha[128];   // Caracteres lidos da linha
    char *palavra;     // Ponteiro para a palavra atual
    char *proximo;     // Ponteiro para o caractere após a palavra atual

```

Fonte: os autores.

Figura 6 - Estruturas condicionais

```
21      // Enquanto houver linhas para ler
22      while ((proximo = fgets(linha, sizeof(linha), entrada)) != NULL) {
23          // Enquanto houver palavras na linha
24          while ((palavra = strpbrk(proximo, LETRAS)) != NULL) {
25              // Encontra o final da palavra
26              proximo = palavra + strspn(palavra, LETRAS);
27
28              // Termina a palavra com o caractere nulo
29              if (*proximo != '\0') { *proximo++ = '\0'; }
30
31              // Ignora as palavras com uma única letra
32              if (strlen(palavra) == 1) { continue; }
33
34              // Transforma a palavra em letras minúsculas
35              strtolower(palavra);
36
37              // Busca a palavra na tabela hash
38              Registro *registro = Tabela_buscar(tabela, palavra);
39
40              // Adiciona a linha atual à lista de linhas da palavra
41              registro->linhas[registro->ocorrencias++] = numLinha;
42          }
43
44          // Incrementa o contador de linhas
45          numLinha++;
46      }
```

Fonte: os autores

2.2.3 FUNÇÃO IMPRIMIR OCORRÊNCIAS

A função *imprimir_ocorrencias* lê o arquivo de pesquisa, faz o tratamento de caracteres, transformado maiúsculo e minúsculo, contando os caracteres e linhas a serem percorridas.

Após padronizar os registros a função imprime os dados de saída que estavam armazenados na hash, veja na figura 7:

Figura 7 - Função *imprimir_ocorrencias*

```c
void imprimir_ocorrencias(Tabela *tabela, FILE *pesquisa) {
    int  numPalavras = 0; // Número de palavras a serem pesquisadas
    char linha[128];      // Caracteres lidos da linha

    // Lê a quantidade de palavras a serem pesquisadas
    fscanf(pesquisa, "%d%*c", &numPalavras);

    // Enquanto houver palavras para pesquisar
    while (numPalavras-- > 0) {
        // Lê a palavra a ser pesquisada
        fgets(linha, sizeof(linha), pesquisa);

        // Remove o caractere de nova linha do final da palavra
        linha[strcspn(linha, "\n")] = '\0';

        // Transforma a palavra em letras minúsculas
        strtolower(linha);

        // Busca a palavra na tabela hash
        Registro *registro = Tabela_buscar(tabela, linha);

        // Mostra a palavra e quantas vezes ela ocorre
        printf("%d %s", registro->ocorrencias, linha);

        // Mosta as linhas em que a palavra aparece
        for (int i = 0; i < registro->ocorrencias; i++) {
            // Não repete a linha se ela já foi mostrada
            if (i == 0 || registro->linhas[i] != registro->linhas[i - 1]) {
                printf(" %d", registro->linhas[i]);
            }
        }
        printf("\n");
```

Fonte: os autores

3 TESTES

Para tratar erros e analisarmos a eficiência do algoritmo, realizamos testes com 3 arquivos, os disponibilizados pelo professor, um arquivo obtido da internet e um que em tese violaria as especificações de número de linhas e caracteres definidos nas especificações do sistema.

3.1 TESTE 1

Teste realizado com os arquivos *input.txt* e *pesquisa.txt,* disponibilizados pelo professor:

Figura 8 - Teste 1

```
lucasferst@LAPTOP-AC8BN78T:/mnt/c/Users/lucas/D
/lucas/Desktop/LucasFerreira_RianWagner
$ make
gcc -o prog main.c analisador.c hashtable.c
lucasferst@LAPTOP-AC8BN78T:/mnt/c/Users/lucas/D
esktop/LucasFerreira_RianWagner$ ./prog ./teste
s/input.txt testes/pesquisa.txt
5 lagarto 1 2 3
5 papel 1 2 3 4
5 pedra 1 2 4
5 spock 1 3 4
5 tesoura 1 2 3 4
2 esmaga 2 3
```

Fonte: os autores.

Nesse teste, o algoritmo mostrou-se bem eficaz, tendo o resultado obtido igual ao esperado nas especificações. Tal fato pode ser observado comparando as figuras 8 e 9.

Figura 9 - Resultado esperado teste 1.

Saída no terminal

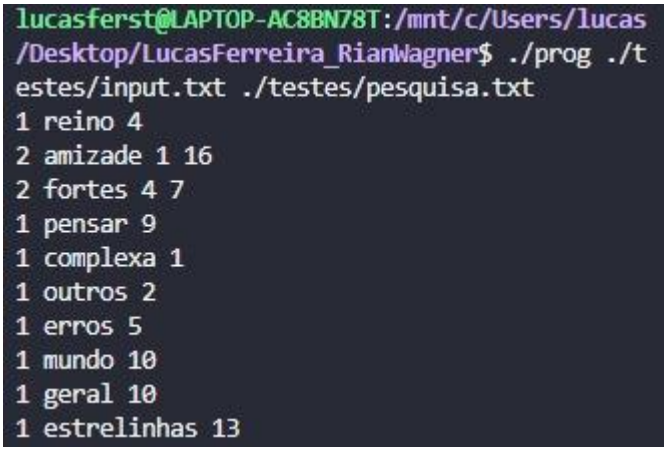
```
5 lagarto 1 2 3
5 papel 1 2 3 4
5 pedra 1 2 4
5 spock 1 3 4
5 tesoura 1 2 3 4
2 esmaga 2 3
```

Fonte: Especificações do trabalho 1 de AEDS II

3.2 TESTE 2

Teste realizado com um arquivo aleatório encontrado na internet. Texto com 20 linhas e pouca repetição de palavras

Figura 10 - Teste 2



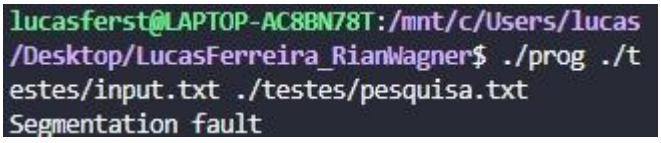
Fonte: os autores.

Nesse segundo teste, o algoritmo mostrou-se eficiente e entregou o resultado esperado. Logo, concluímos que, dentro das especificações de número e tipo de caractere e número de linhas, o programa cumpre o seu papel para um texto comum.

3.3 TESTE 3

Teste realizado em um .txt que extrapola os limites de caracteres por linhas, de palavras por linhas, e de palavras distintas.

Figura 11 - Teste 3



Fonte: os autores.

4 CONCLUSÃO

Tendo surgido no princípio dos problemas de contagem, a ideia de dividir ou “fazer picadinho” de um conjunto de elementos, facilitou tarefas simples. Na computação, essa ideia virou estrutura, a hash. Neste trabalho foi possível concluir como a função de hashing funciona e ter um contato maior com sua aplicabilidade.

Por meio das análises dos testes, concluímos que a estrutura de hash é simples, econômica e muito viável para problemas simples e sem uma grande especificação de elementos.

Nessa perspectiva, tivemos a oportunidade de fazer análises de outras formas de função de hashing e outras estruturas, como hash encadeada, que não foi utilizada no trabalho, mas cogitamos e analisamos suas aplicações.

Por fim, podemos considerar o trabalho como exitoso, o mesmo se mostrou aplicável para situações distintas, como era definido inicialmente.

4 REFERÊNCIAS

[1] https://files.cercomp.ufg.br/weby/up/498/o/Danilo2009.pdf

[2] http://www.ijecs.in/index.php/ijecs/article/view/651/580

[3] http://www.cse.yorku.ca/~oz/hash.html

[4] https://acervolima.com/aplicacoes-de-hashing/

[5] https://blog.pantuza.com/artigos/tipos-abstratos-de-dados-tabela-hash