Trabalho de Spell Checking (Estrutura de Dados)

Matheus de Souza Barroso Ambrosio
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
matheusambrosio@eng.ci.ufpb.br

Francisco Lima Cavalcante Filho
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
franf964@gmail.com

Lucas Henrique Lopes Rodrigues da Silva
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
lucasrodrigues@eng.ci.ufpb.br

I. Resumo

Este artigo fala sobre o desenvolvimento de um sistema de dicionário com o uso de tabela *hash* para a linguagem C. Foram criadas funções para manipulação de arquivos, e, no fim, foi estudada a relação de número de *buckets* x tempo de execução, com o intuito de encontrar um bom equilíbrio entre eficiência e uso de memória. Além disso, foi realizada uma comparação entre a função implementada neste artigo, com a função de *hash* padrão da linguagem Python.

II. Introdução

Spell Checking (Checagem de Escrita, em tradução livre), consiste em um algoritmo de leitura e verificação de texto, através de um banco de dados formado por diversas palavras. Neste artigo, falaremos sobre a implementação de um *Spell Checking* na linguagem C, através do uso da estrutura de Tabela *Hash*.

Para a realização desse trabalho, foi utilizada uma função de *hash* chamada sdbm, as colisões nos *buckets* foram tratadas pelo método de *separate chaining*, e o texto corrigido foi "O Alcorão Sagrado".

III. As Funções

O código é dividido em três partes principais: o carregamento do dicionário, a verificação do arquivo a ser corrigido, e a criação do relatório de saída, além da função de *Hash*.

1 – Carregamento do Dicionário

Na função LoadFile() (Figura 1), é realizada a leitura de cada palavra presente no arquivo de dicionário, onde, para cada palavra, é calculado um valor de *hash*, e, de acordo com este valor, a palavra é alocada em um dos *buckets* da tabela.

```
while(fscanf(archive, "%s", input) != EOF){

    Node* node = (Node*)malloc(sizeof(Node));
    strcpy(node->word, input);
    node->next = NULL;

    bkt = HashFunction(input);

    node->next = hashTable[bkt];
    hashTable[bkt] = node;
}
```

Figura 1 – Trecho da Função "LoadFile()", que preenche a tabela *hash.*

2 – Verificação do arquivo

A verificação do arquivo é feita pela função ReadFile() (Figura 2), onde o texto do arquivo é armazena do em uma variável, que depois é dividida a cada palavra, onde esta, por vez, tem seu valor de *hash* calculado, e é verificado se a palavra está presente do seu *bucket* correspondente. Se não, é adicionada à lista de palavras escritas incorretamente.

```
countWords++;
found = 0;
bkt = HashFunction(input);
aux = hashTable[bkt];

while(aux != NULL){

    if(!strcmp(aux->word, input)){
        found = 1;
        break;
    }

    aux = aux->next;
}
```

Figura 2 – Trecho da Função "ReadFile()", que verifica a escrita das palavras.

3 – Relatório de Saída

A função Conclusion() é responsável por criar o arquivo conclusion.txt (Figura 3), no qual se encontra um *feedback* do que ocorreu durante a execução do *Spell Checking*. Este arquivo contém a quantidade de palavras analisadas, o tempo gasto com a análise, e a lista de palavras que falharam.

```
Verification time: 116.00 ms
Number of words in the input text: 305950
Number of miswritten words in the input text: 150194
List of miswritten words:

-org
-eBooksBrasil
-2006
```

Figura 3 – Parte do arquivo “conclusion

4 – Função de *Hash*

Durante o desenvolvimento desse algoritmo, foi utilizada a função de *hash* sdbm (Figura 4), recomendada por sites que trabalham com a área. Na função, passamos a palavra, e, através de operações matemáticas complexas, é gerado um número, que, após ser dividido pelo número de *buckets*, se torna o índice da palavra na tabela *hash*.

```
unsigned int HashFunction(unsigned char *word){

    unsigned long hash = 0;
        int c;

        while (c = *word++)
            hash = c + (hash << 6) + (hash << 16) - hash;

        return hash%NBUCKETS;
}
```

Figura 4 – Função “HashFunction()” na qual se calcula o valor de *hash* de cada palavra.

IV. Estudo de Casos

1 – Quantidade de *Buckets*

Após a implementação das funções, foram realizados testes e estudos, visando descobrir a melhor combinação entre baixo consumo de memória e alta velocidade de processamento. Além disso, também foi levado em conta o balanceamento dos *buckets*, ou seja, a variação na quantidade de elementos em cada *bucket*.

Núm. De Buckets	Tempo	Num. de Buckets Vazios
300000	83	107495
200000	86	42950
150000	91	19236
120000	95	9371
100000	99	4617
75000	107	1234
50000	132	124
30000	182	1
10000	359	0
5000	701	0

Tabela 1 – Relação de *buckets* utilizados, tempo de leitura e *buckets* vazios (teste realizado utilizando o próprio dicionário como texto de entrada).

Analisando a tabela 1, pode-se perceber que, ao aumentar a quantidade de *buckets* utilizados, o tempo de leitura diminui, porém, a quantidade de *buckets* desperdiçados aumenta consideravelmente. Tendo isso em vista, elaborou-se o gráfico 1, com o intuito de descobrir o número de *buckets* com maior aproveitamento de memória e tempo de leitura.

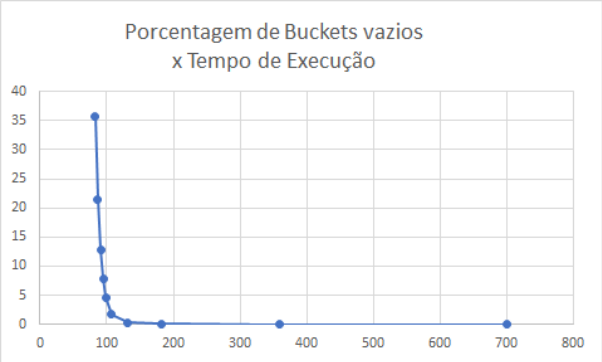

Gráfico 1 - Taxa de *buckets* desperdiçados por tempo de leitura.

Analisando o Gráfico 1, podemos perceber que a área que nos interessa é a área mais próxima à origem (0, 0). Com isso em mente, o ponto mais adequado para se utilizar é o ponto que representa 75000 buckets. Este ponto, de acordo com a tabela 1, apresenta um consumo de memória consideravelmente baixo, assim como seu tempo de leitura.

Ao realizar os testes utilizando de fato o arquivo de entrada, e não o próprio dicionário, foi constatado que o *spell checking* foi realizado em 93 ms, e um total de 1234 *buckets* vazios, como esperado devido aos testes anteriores.

2 – Tratamento de Colisões

O tratamento de colisões foi feito utilizando o método *separate chaining*, que consiste em, quando um elemento for ocupar uma determinada posição na tabela *hash*, e tal posição já estiver ocupada por algum outro elemento, este será adicionado em uma outra posição de uma fila, porém, no mesmo *bucket*.

Com finalidade de acelerar o processo de inserção na tabela *hash*, foi feito um esquema, onde, ao se adicionar um novo elemento à fila, este é inserido no começo da mesma, e não no fim. Com isto, o processo de inserção realiza menos comandos, assim, sendo executado em menor tempo.

V. Comparação com outra Linguagem

1 - Código

Foi criado um programa em Python (Figura 5), com funcionamento similar ao desenvolvido na linguagem C, porém, utilizando a função de *hash* padrão da linguagem Python, com a intenção de comparar seu tempo de leitura com o do código implementado neste projeto.

```
def Search(testFile,dicti):
        testFile = open(testFile,"r")
        resultFile = open("conclusion.txt","w")

        linha2 = testFile.readline()
        timer1 = time.time() * 1000
        while linha2:
                text = linha2.split()
                for string in text:
                        if string not in dicti:
                                resultFile.write(string + "\n")

                linha2 = testFile.readline()

        timer2 = time.time() * 1000
        print(timer2-timer1)
        testFile.close()
        resultFile.close()
```

Figura 5 – Trecho do código em Python, responsável pela busca de palavras na *hash*.

2 - Resultado

Após a implementação, foi constatado que o tempo de leitura do programa em Python foi de 270 ms, enquanto no em C, 93. Isto se deve ao fato de que a função de *hash* de Python é genérica, ou seja, serve para organizar qualquer tipo de dados em uma tabela *hash*, enquanto a implementada em C é focada em uso de dicionário, com instruções específicas otimizam seu tempo de execução.

VI. Referências

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

http://www.ebooksbrasil.org/adobeebook/alcorao.pdf

https://stackoverflow.com/questions/3002122/fastest-file-reading-in-c