Relatório do Trabalho sobre Spell Checking

Bruno Henrique Araújo da Costa
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
brunocosta@eng.ci.ufpb.br

Gabriel Lucena Camboim
Universidade Federal da Paraíba
Centro de Informática
João Pessoa, Paraíba, Brasil
gabrielcamboim@eng.ci.ufpb.br

I. Introdução

O trabalho consiste na elaboração de um *spellchecker* por meio da utilização de uma tabela *hash*. Um *spellchecker* trata-se de uma espécie de "verificador de palavras", o qual compara palavras recebidas com as palavras de um dicionário, as quais estão armazenadas em uma tabela *hash* previamente carregada pelo programa.

Implementou-se uma tabela *hash* utilizando uma função de *hash* já existente e tratamos as suas colisões por meio de encadeamento separado (*separate chaining*) com a implementação de vetores de listas encadeadas para armazenar palavras em um mesmo *bucket*. A função de *hash* utilizada foi escolhida com base em recomendações de diversas comunidades virtuais e após sucessivos testes de desempenho entre algumas funções escolhidas.

II. Desenvolvimento

```
while(fscanf(dicio, "%s", buffer) > 0) {
    node* no = malloc(sizeof(node));
    no->next=NULL;

    strcpy(no->word, buffer);

    bucket = Hash_Function(no->word);

    if(hash_table[bucket]->word == NULL) {
        hash_table[bucket]=no;
    } else {
        no->next=hash_table[bucket];
        hash_table[bucket]=no;
    }
}
```

Fig. 1. Trecho de código referente ao carregamento do dicionário

Inicialmente, o arquivo do dicionário é carregado no programa e a leitura é feita palavra por palavra. Cada palavra passa pela função de *hash* que retorna seu valor *hash*, em seguida a palavra é adicionada na tabela em seu *bucket* específico. Para tratar as colisões, utilizamos uma lista simplesmente encadeada que fica responsável por organizar todas as palavras localizadas em um mesmo *bucket*. Dessa forma, após a finalização da função, todas as palavras ficam organizadas em vetores de *linked lists* para facilitar o acesso.

```
while(!feof(text)) {

    memset(buffer, BUFFER_FILE_LINE, '\n');
    if(!fgets(buffer, BUFFER_FILE_LINE, text)) break;

    CmpTxt = strtok(buffer, SPECIALCHARACTERS);

    while(CmpTxt!=NULL){
        counterHowManyWords++;

        if(verify(CmpTxt)==0) {
            counterNotFoundWords++;
            node* no = malloc(sizeof(node));
            no->next=NULL;

            strcpy(no->word, CmpTxt);
            if(notFoundWords == NULL) {
                notFoundWords=no;
            } else {
                no->next=notFoundWords;
                notFoundWords=no;
            }
            no = NULL;
            free(no);
        }
    CmpTxt = strtok(NULL, SPECIALCHARACTERS);
    }
}
```

Fig. 2. Trecho de código referente à leitura do arquivo de texto

Após o término do carregamento do dicionário, o processo de tratamento do arquivo de texto recebido é iniciado. Diferentemente do que ocorre no dicionário, no arquivo em questão a leitura é feita linha por linha, dentro de um laço de repetição o qual possibilitará a divisão da linha em diversas *strings*, juntamente com a retirada de caracteres especiais previamente definidos no início do código por meio da função strtok() da biblioteca <string.h>.

```
callFunction = Hash_Function(buffer);

aux = hash_table[callFunction];
while(aux!=NULL) {

    if(strcmp(buffer, aux->word)==0) {
        return 1;
    }

    aux = aux->next;
}
```

Fig. 3. Trecho de código referente à função verify

Com o auxílio da função *verify*, é verificado se cada *string* está localizada na lista encadeada do *bucket* correspondente ao seu valor *hash*. Considerando que a palavra não seja localizada, esta é adicionada em uma lista encadeada contendo todas as outras palavras que também não foram encontradas. Além disso, foi implementado com auxílio da biblioteca <time.h>, um mecanismo que armazenará o tempo que a função de leitura do arquivo de texto levou para ser executada.

III. Estudo de caso

As variáveis possíveis para denotar o desempenho da tabela *hash* são: o método de tratamento de colisões, quantidade de *buckets* utilizados, dispersão dos elementos na tabela *hash* e também a função de *hash* empregada. Tais variáveis serão discorridas a seguir.

A. Método de tratamento de colisões

```
typedef struct node {
    char word[WORD_SIZE];
    struct node* next;
} node;
```

Fig. 4. Trecho de código referente à criação de um nó utilizado em listas encadeadas

Como visto durante as aulas, os dois possíveis meios de tratamento de colisões são o *Separate Chaining* e *Linear Probing*. Foi observado que o método de *Separate Chaining* possui um desempenho bastante similar em funções de hash diferentes, enquanto o método de *Linear Probing* pode sofrer de uma redução significativa na performance da tabela se uma função de *hash* inadequada for escolhida.

Além disso, constata-se que para uma quantidade de chaves maior, o *separate chaining* mostra-se mais eficiente devido ao fato de não ter que percorrer centenas ou milhares de *buckets* em busca da chave correspondente. Assim, optou-se por utilizar o método de separate chaining, buscando atrelar eficiência à estabilidade.

B. Quantidade de buckets

Uma variável indispensável para o programa trata-se da quantidade de *buckets*. Observa-se que a quantidade de *buckets* é diretamente proporcional à rapidez com que o programa é executado, porém, somente até certo ponto. Faz-se necessário também otimizar o consumo de memória, pois uma vez que há um aumento no número de *buckets*, também haverá um maior consumo de espaço de armazenamento.

Para a realização dos testes, foi analisada a Constituição da República Federativa do Brasil de 1988 e para cada valor nos pontos indicados no gráfico, foram realizados 10 (dez) testes individuais de tempo de execução da função de checagem, em seguida foi realizada uma média aritmética com base nos resultados obtidos.

A seguir pode-se ver os testes realizados com alguns valores de *buckets*:

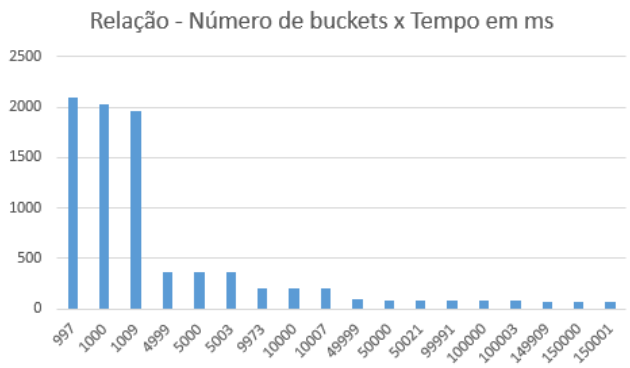

Fig.5 . Gráfico: buckets x tempo gasto em milissegundos

Com o teste de 18 (dezoito) valores de buckets diferentes, incluindo valores primos, observa-se através do gráfico que a relação custo-benefício entre o número de *buckets* e o tempo de execução passa a ser inviável a partir de 49.999 *buckets*, ficando mais evidente ainda a partir de 50.000, pois o custo de memória necessário para se obter um ganho notório no desempenho é muito alto. Assim, foi escolhido os 50.000 *buckets* com base nos dados apresentados.

C. Dispersão dos elementos na tabela hash

Uma boa função de *hash* está relacionada ao bom uso dos espaços designados para armazenamento dos elementos. Uma distribuição homogênea garante estabilidade no processo de procura, além de minimizar parcialmente seu tempo.

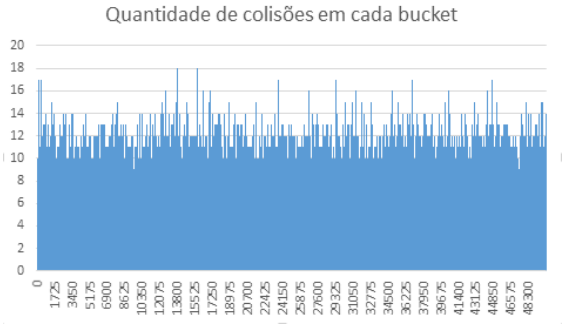

Fig. 6. Gráfico: Quantidade de colisões em cada bucket

Como é possível notar no gráfico acima, a função de *hash* selecionada garante que grande parte dos *buckets* serão preenchidos, resultando em um melhor gerenciamento da memória disponível.

```
unsigned int Hash_Function(const char* word) {
    int size = strlen(word);
    int i=0;
    unsigned int value = 5381;

    for(i = 0; i < size; ++i) {
        value += (value << 5)  + word[i];
    }

    return value % BUCKETS;
}
```

Fig. 7. Código referente à função de hash

A função de *hash* escolhida funciona em torno de um número primo que está distante de potências de dois. Para cada caractere da palavra, é incrementado ao número primo (*value*) o seu valor com 5 (cinco) bits deslocados à esquerda, somados com o valor ASCII do caractere em questão. Após o incremento, é retornado o valor *hash* com base no resto da divisão entre o valor com o incremento e o número de *buckets*.

Baseado em recomendações de sites que mostraram a utilização de funções de *hash* para distribuição de *strings* em uma tabela, a função em questão foi escolhida, além disso testamos diversas funções que tinham como mesmo objetivo dispersar *strings* em uma tabela de distribuição e selecionamos a que obteve um ligeiro melhor desempenho.

IV. Extra

Para testar a eficiência do código apresentado, foi utilizada uma tabela *hash* já implementada por meio da linguagem Python. Ela funciona de maneira similar, porém utilizando uma função de *has*h genérica para atender a qualquer necessidade.

Considerando que a tabela *hash* já se encontra pronta, apenas buscou-se implementar a função de checagem da palavra no dicionário, que funciona de maneira ligeiramente diferente, mas com as mesmas intenções.

```
def Searching(self, fileToCompareWithDictionary):

    counterHowManyWords = 0
    counterNotFoundWords = 1 # Seta o índice em 1
    text = open(fileToCompareWithDictionary, "r")
    finalReport = open("results.txt", "w")

    fileLine = text.readline()
    start = time.time()

    while fileLine:
        line = fileLine.split()
        for string in line:
            counterHowManyWords += 1
            if string not in self.Dictonary:
                finalReport.write('%d' % counterNotFoundWords + "- " + string+"\n")
                counterNotFoundWords += 1
        fileLine = text.readline()
    text.close()
```

Fig. 8. Código referente à pesquisa (Python)

Inicialmente é realizada a abertura do arquivo contendo o texto a ser comparado e sua leitura é feita linha por linha. Em um laço de repetição, a linha é dividida em palavras e cada uma delas é checada no dicionário, incrementando um contador caso a palavra não seja localizada. Após tal, dados estatísticos referentes à função são inseridos em um arquivo de texto.

V. Resultados

Os resultados obtidos a partir do dicionário disponibilizado pelo professor foram de 428 milissegundos em Python e de 266 milissegundos para a implementação em C, utilizando como texto a ser analisado o próprio dicionário fornecido.

```
Número total de palavras do texto: 307855
Número de palavras não encontradas no texto: 0
Tempo total da verificação: 428 ms
```

Fig. 9. Dados referentes à execução (Python)

```
Número total de palavras do texto: 307855
Tempo total da verificação: 266 ms
Número de palavras que falharam no spell check: 0
```

Fig. 10. Dados referentes à execução (C)

A comparação revela que o uso de uma função de *hash* específica para determinada atividade a confere um ganho de desempenho (chegando a ser cerca de metade do tempo de uma genérica), além de outras variáveis como tratamento de colisões e memória dedicada.

VI. Referências

HASHING I SET 2 (SEPARATE CHAINING). Geeks for Geeks. Disponível em: <www.geeksforgeeks.org/hashing-set-2-separate-chaining>. Acesso em: 30 ago. 2019.

HASH MAPS WITH LINEAR PROBING AND SEPARATE CHAINING. I Need Coffee. Disponível em: <https://ineed.coffee/44/hash-maps-with-linear-probing-and-separate-chaining/>. Acesso em: 30 ago. 2019.

HASH TABLE IMPLEMENTATION IN PYTHON. Mukesh Chapagain Blog. Disponível em: <http://blog.chapagain.com.np/hash-table-implementation-in-python-data-structures-algorithms/>. Acesso em: 30 ago. 2019.

CONSTITUIÇÃO DA REPÚBLICA FEDERATIVA DO BRASIL DE 1988. Planalto.gov.br. Disponível em: <http://www.planalto.gov.br/ccivil_03/constituicao/constituicao.htm>. Acesso em: 30 ago. 2019.