Implementação de Spell Checking utilizando tabela hash

Elaine Cristina Lins de Souza
Universidade Federal da Paraíba
João Pessoa - Paraíba, Brasil.
Caixa Postal 58055-000
Email: elainesouza@eng.ci.ufpb.br

Rebeca Raab Bias Ramos
Universidade Federal da Paraíba
João Pessoa - Paraíba, Brasil.
Caixa Postal 58055-000
Email: rebecabias@eng.ci.ufpb.br

Thiago Marques Silva
Universidade Federal da Paraíba
João Pessoa - Paraíba, Brasil.
Caixa Postal 58055-000
Email: thiagomarques@eng.ci.ufpb.br

***Resumo*—O presente relatório apresenta o processo de criação de um spell checking através de uma tabela hash, cujo objetivo é aprimorar os conhecimentos aprendidos em sala de aula e fazer uma avaliação e comparações sobre as dificuldades encontradas na tentativa de otimizar o desempenho do programa.**

I. Introdução

Os métodos de organização de dados surgiram com o propósito de resolver um dos principais problemas do âmbito da programação, e embora esses métodos tenham evoluído, o gerenciamento e aplicação de estruturas de dados para a construção de um bom algoritmo ainda requer uma certa quantidade de cautela, uma vez que cada método possui funcionalidades diferentes. Para a construção de um Spell Checking, programa de software ou recurso de programa projetado para localizar palavras e notificar o usuário sobre os erros ortográficos (HOPE, 2018) [1], a tabela hash ou tabela de dispersão torna-se uma ótima ferramenta, uma vez que é utilizada para mapear um determinado valor com uma chave específica simples que serve para tornar o acesso aos elementos mais rápido.

A eficácia do mapeamento depende da eficiência da função de hash usada. Em síntese, a função converte um identificador (chave) que pertencente a algum domínio relativamente grande em um número de um intervalo relativamente pequeno (BENTO, SÁ, SZWARCFITER - 2015) [2], isso tende a facilitar quando o objetivo principal do programa é a busca por dados. Em outras palavras, a função de hash executa a transformação do valor de uma chave em endereço, pela aplicação de operações aritméticas e/ou lógicas. Além disso, ela deve espalhar as chaves pelo intervalo de índices de maneira linear para ser considerada ideal, e apresentar algum método de tratamento de colisão como o Open Addressing, que baseia-se na ocupação de um elemento por bucket alocando o próximo elemento no próximo slot disponível no caso de haver colisão ou se já foi alocado algum elemento para posição, e o Separate Chaining, que consiste em fazer com que cada célula da tabela de hash aponte para uma lista encadeada de registros.

Dessa forma, este trabalho possui como principal objetivo o desenvolvimento de um algoritmo, utilizando tabela hash e outras estruturas de dados auxiliares, que busca e mapeia um arquivo com 307856 palavras e posteriormente faz a comparação com um arquivo de entrada contendo palavras para verificação.

II. Metodologia

O desenvolvimento do programa se deu através de várias etapas que foram fundamentais para a conclusão do projeto, as etapas se constituíram em:

1 - Planejamento: Essa fase consistiu no levantamento prévio dos recursos a serem utilizados para a construção do programa em geral e da construção da lógica de programação necessária para implementação da tabela hash. Foi escolhido também o método separate chaining para o tratamento de colisões.

2 - Execução: Nessa etapa nós começamos a colocar em prática tudo aquilo que tínhamos planejados na etapa anterior. O programa então foi feito na linguagem C, através da IDE "DEV C++" na versão 5.11.

3 - Testes: Uma vez feito o programa, começamos a testar funções hash diferentes para a verificação de qual delas se adaptaria melhor à nossa implementação, além de escolher quantos buckets seriam necessários para um menor número de colisões. Escolhemos o número 59388 para ser a quantidade de buckets e também 59387 (primo) como número para fazer o MOD. Os testes foram realizados em um notebook Dell® Inspiron modelo i15-3576 A70C.

4 - Análise dos resultados: Foi feita uma avaliação de qual das funções hash obtinham um maior desempenho e o porquê disso.

5 - Construção do relatório: Etapa de conclusão do projeto, constituindo-se em apresentar toda a metodologia e explicações sobre o que foi utilizado para a construção do programa.

III. Desenvolvimento

No primeiro caso de teste, foi feita uma função que recebia uma string e o seu tamanho, onde posteriormente era somado o valor da tabela ASCII de cada letra. Após isso, era feito o mod com um número primo escolhido arbitrariamente e, após ser calculado o hash, a palavra era encaminhada ao seu respectivo bucket. Isso resultou em um algoritmo que não dependia do número de buckets, ou seja, a distribuição ocorria unicamente pelo valor do número que era feito o

mod. Como consequência, tínhamos uma distribuição muito abaixo do esperado, visto que a função não fazia uma relação do número de buckets com as alterações necessárias para espalhar os dados da maneira mais randômica possível. Nessa função, os dados se concentravam em um ponto específico, não importando a quantidade de buckets que nós colocássemos.

```
unsigned int hashFunction(char *word, int maxSize){

    unsigned int i = 0, sum = 0;

    for(i = 0; i < maxSize; i++){

        sum += word[i];
    }

    return sum % NUMBER_OF_MOD;
}
```

Figura 1. Primeira função de hash utilizada no algoritmo.

Na tentativa de construir uma função hash que melhor se adaptasse ao nosso objetivo, foi utilizado o algoritmo da função "One at a time hash" na segunda e atual versão, que foi desenvolvido por Bob Jenkins, com base nos requisitos de Colin Plumb (um criptógrafo que supostamente trabalha para a NSA) e consiste em produzir um valor de 4 bytes a partir de uma entrada de vários bytes (GUIDES, 2016, tradução livre) [3].

Assim essa função busca executar operações mais simples que garantem que no final os bits nos últimos bytes sirvam como um "avalanche" para os bytes de saída, sendo o avalanche a probabilidade que um bit será alternado se o bit de entrada for invertido. (HSIEH, 2008, tradução livre) [4]. Em outras palavras a função recebe uma string do usuário e soma o valor da tabela ASCII de cada letra. Posteriormente, são feitos alguns deslocamentos de bits e operações lógicas para a geração da chave final.

Com isso, a função apresentou uma boa distribuição por Separate Chaining, pois diferente da função anterior, essa já conseguia gerar um ruído melhor na chave, mesmo que duas palavras tivessem o mesmo hash inicial, dado que a função parte da premissa da soma do valor da tabela ASCII de cada letra. Uma das principais características dessa função é que a distribuição está ligada diretamente ao número para fazer o mod e a quantidade de buckets, já que quanto menor o número, mais concentrado em uma pequena parcela de buckets as chaves vão ser geradas, enquanto que se aumentarmos demais o número de buckets, o hash começará a deixar muitos buckets vazios.

Após algumas adaptações necessárias para o melhor funcionamento do algoritmo, como a aplicação do mod na chave gerada pela função de hash com o objetivo de fixá-las no intervalo fechado de [0, 59387], o algoritmo da função de hash ficou:

```
unsigned int hashFunction(char *word, int maxSize){

    unsigned int i = 0, hash = 0;

    for(i = 0; i < maxSize; i++){

        hash += word[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return (hash % NUMBER_OF_MOD);
}
```

Figura 2. Função de hash atual utilizada no algoritmo.

Quanto ao desempenho das duas funções ao executar a pesquisa da constituição, a função one at a time apresentou um desempenho muito melhor do que a primeira função empregada no projeto para uma entrada com 6983 palavras. A primeira função levou 100ms $\pm$ 2ms, enquanto a função one at a time levou 5ms $\pm$ 1ms.

IV. Conclusão

Diantes de todos os testes e estudos que realizamos, concluímos que uma boa função de hash depende de um equilíbrio entre: número de buckets e geração de ruído na chave. Além disso, a relação entre a quantidade de buckets e o valor do mod, consiste normalmente em uma potência de 2 para o número de buckets e o número primo menor e mais próximo dessa potência para a o valor do mod.

Percebemos ainda, que nossa função não irá mapear o último bucket, pois sempre usamos um número menor que o de buckets para o mod e que não importa o quanto tentássemos manipular a quantidade de buckets ou o número do mod, se o problema estiver na geração da chave na função de hash, o máximo que irá acontecer é a concentração das chaves em um determinado ponto, implicando em uma péssima distribuição.

Portanto, embora a construção de uma função de hash perfeita seja um trabalho muito complexo, aproximações através de uma boa função hash são totalmente possíveis, provando assim que uma simples função de hash como a nossa pode obter um bom resultado em relação à quantidade de memória e em tempo de execução.

Referências

[1] *HOPE, Computer. Spell Checker. 2018.* Acesso em: 02 Set. 2019. [Online]. Available: https://www.computerhope.com/jargon/s/spelchec.htm

[2] *BENTO, Lucila Maria de Souza; SA, Vinícius Gusmão Pereira de; SZWARCFITER, Jayme Luiz. SOME ILLUSTRATIVE EXAMPLES ON THE USE OF HASH TABLES. Agosto de 2015.* Acesso em: 01 Set. 2019. [Online]. Available: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382015000200423&lng=en&nrm=iso.

[3] *GUIDES, Open Tech. Implementing Jenkins one-at-a-time hash function in VB.NET. 2018.* Acesso em: 02 Set. 2019. [Online]. Available: https://www.opentechguides.com/how-to/article/vb/173/jenkins-hash.html

[4] *HSIEH, Paul. Hash functions. 2008.* Acesso em: 02 Set. 2019. [Online]. Available: http://www.azillionmonkeys.com/qed/hash.html