Implementação e Análise de Funções de *Hash* no Desenvolvimento de um Programa *Spell Checker*

Laura Letícia A. O. Campos[1], Stênio Ellison P. Ferreira[2], Wendson Carlos S. Silva[3]
Centro de Informática
Universidade Federal da Paraíba - UFPB, João Pessoa-PB, Brasil
llauracampos@gmail.com[1], stenio1998@gmail.com[2], wendsoncarlos09@gmail.com[3]

***Resumo*—Este artigo tem como proposta implementar um programa *Spell Checker* e analisar as funções de espalhamento mais eficientes para a correção de palavras. São apresentadas funções de *hash* extraídas da literatura que possuem desempenho satisfatório. Os resultados revelam uma homogeneidade na performance, com enfâse ao *DJB2* que apresenta algumas melhorias diante os outros métodos.**

***Palavras-Chave*: corretor ortográfico, função de *hash*, otimização;**

IMPLEMENTATION AND ANALYSIS OF HASH FUNCTIONS IN THE DEVELOPMENT OF A SPELL CHECKER PROGRAM

***Abstract*—This article aims to implement a Spell Checker program to analyze the most efficient hash functions for word correction. Hash functions extracted from the literature that have satisfactory performance are presented. The results reveal a homogeneity in performance, with emphasis on DJB2 that presents some improvements over other methods.**

***Keywords*: spell checker, hash function, optimization;**

I. Introdução

A ciência e a tecnologia são elementos primordiais para a evolução da sociedade como um todo, tendo em vista que, sem a integração entre eles seria pouco provável que se atingisse algum grau de desenvolvimento. Inserida nesse cenário, a computação é uma área em constante ascensão desde a sua criação, na primeira metade do século XX, e, desde então, buscam-se soluções de otimização para as mais diversas funcionalidades de um computador.

Nesse contexto, visando a necessidade de fazer a correção de textos de forma eficiente e prática, cientistas da Universidade de Stanford desenvolveram, no início da década de 70, uma forma de computadores verificarem a ortografia de palavras contidas em um arquivo. Esse processo foi chamado de *Spell Checker*, um programa que examina um texto, extrai as palavras contidas nele e, em seguida, compara cada termo com uma lista conhecida de palavras corretamente escritas, ou seja, um dicionário. Ao longo das décadas, com o intuito de aperfeiçoar esse processo, foram elaboradas diversas formas de implementar essas listas, com diferentes tipos de estruturas de dados, entre elas, a função de espalhamento, também conhecida como função de *hash*.

Segundo Cormen (2002, p.179)

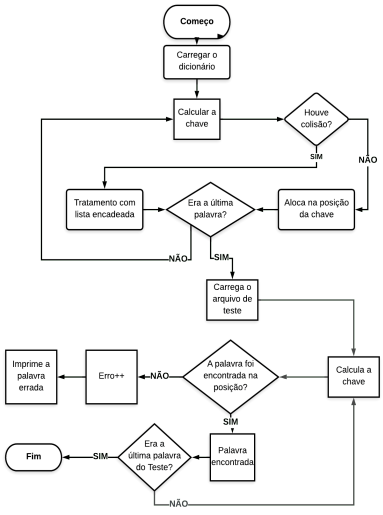
> Uma tabela *hash* é uma estrutura de dados eficiente para implementar dicionários. Embora a busca por um elemento em uma tabela *hash* possa demorar tanto quanto procurar por um elemento em uma lista ligada - o tempo O (n) no pior caso – na prática, o *hash* funciona extremamente bem. Sob hipóteses razoáveis, o tempo esperado para a busca por um elemento em uma tabela *hash* é O (1).

Sendo assim, este trabalho tem por objetivo apresentar a realização de um programa *Spell Checker* a partir do desenvolvimento de três funções de espalhamento, apresentando um comparativo entre elas para definir as vantagens e desvantagens de cada uma e estipular qual, entre estas, possui melhor eficiência e otimização. O projeto foi implementado na *IDE Visual Studio Code*, utilizando a linguagem C.

II. Metodologia

A figura a seguir exibe o funcionamento do programa sintetizado em diagrama de blocos a fim de facilitar o entedimento da execução do programa.

Figura 1: Diagrama de blocos



A função de espalhamento é uma forma de mapear dados de tamanhos variáveis e torná-los pequenos, com um tamanho padrão. Este meio é utilizado para uma busca em uma base

de dados, pois é permitido a comparação das informações entre si para encontrar o que foi buscado. Entretanto, há uma possibilidade da função criar as mesmas posições para chaves diferentes, também conhecida como colisão, que influencia na eficiência do item procurado. Existem diversas funções de *hash* que podem ser agrupadas na função principal, capazes de atenuar tais colisões, já que esse número pode ser minimizado, mas não pode ser eliminado totalmente. O projeto utiliza três funçoes de *hash* e uma função pré *hash*.

A. One at time Jenkins

É uma função de espalhamento criada por Bob Jenkins. Baseado em um estudo feito pelo criptógrafo Colin Plumb, Jenkins idealizou um modelo de função que produz uma *hash* de 4 bytes a partir de uma entrada de vários bytes.

A função recebe a palavra como entrada, a transforma em uma variável do tipo inteiro e é feito o deslocamento de bits percorrendo toda palavra. Para este projeto foi necessária a criação de uma função pré *hash*, que converte as variáveis do tipo *char* para o tipo inteiro, a fim de permitir o descolamento de bits feito pela *One Time Jenkins*.

B. Hash de Bernstein (DJB2)

Criada pelo professor de Ciência da Computação Daniel Julius Bernstein, a função de espalhamento *DJB2* promove o deslocamento de bits de forma diferente da Jenkins. Esta utiliza um tamanho de chave e um número para cálculo que seguem um determinado padrão. Essa norma, denominada "números mágicos", é atribuída à chave e ao produto do resultado. Mathur et al. (2015) afirma que, devido a sua alta velocidade, simplicidade e boa distribuição, a *DJB2* é um dos algoritmos mais utilizados.

Neste projeto foi utilizada uma chave de valor 5381. Em seguida, foi realizado o produto entre essa chave e o número 33 e, a esse resultado, foi somada uma constante. Esses são os valores conhecidos como "números mágicos", que garantem a eficácia da função, o que foi fundamentado no código, por apresentar um menor número de colisões.

C. Método da divisão

A função de espalhamento modular é uma das mais comumente utilizadas em tabelas *hash*. Essa função propicia o cálculo do tamanho da tabela, denominado módulo de *M*. Cada caractere da palavra inserida é transformado para o tipo inteiro e multiplicado por uma constante. Esse resultado é dividido por *M* e o valor do resto dessa divisão indica em qual *bucket* a entrada será armazenada.

No presente trabalho, a *hash* modular se refere ao uso da *Jenkins* e da *DJB2*. Essas funções criavam chaves demasiadamente grandes, devido à multiplicação de bits, o que inviabilizava o programa. Portanto, houve uma mudança das variáveis para o tipo inteiro e a função *Mod* possibilitou a divisão.

D. Tratamento de colisões

Nesta etapa, buscou-se a abordagem do tratamento de colisões, uma vez que é inevitável o surgimento de conflitos, mesmo possuindo uma excelente função de espalhamento. Utilizou-se o método *separate chaining* (encadeamento separado) no qual é aplicada uma lista encadeada para cada índice *i* que armazena os elementos cuja as chaves espalham para *i*. Quando ocorre uma colisão é alocado memória para a criação de um novo nó que incorporará a chave; a variável próximo do nó passa a ser apontada para a chave atual e a nova chave torna-se o primeiro elemento da lista. A seleção desse método é baseada na vantagem de que os itens não precisam estar em armazenamento contínuo e por permitir o percurso dos itens pela sequência da chave de *hash*.

E. Comparação entre funções de hash nas linguagens C e Python

A linguagem *Python*, diferente da C, possui uma estrutura de dados de dicionário chamada *dict*, que é um tipo especial de tabela *hash*, também conhecida como uma matriz associativa. A implementação do dicionário é composta por um conjunto de valores-chave, no qual cada par mapea a chave para um número. Um dos grandes diferenciais dessa estrutura é a capacidade de poder armazenar valores em índices literais em vez de numerais.

Neste projeto, utilizamos a metodologia da *dict*, que usa um nome literal para a chave e o separa do seu valor por ":". Após a implementação, foi construída uma tabela de desempenho para comparar a melhor função de *hash* do código em C com a que foi executada em Python.

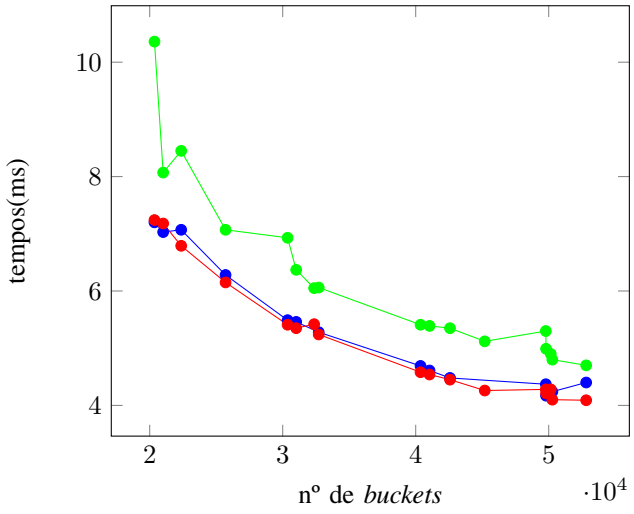
III. Resultados e Discussões

Tabela I: Comparação entre as funções de *hash*

Nº de palavras	**Tempos médio (ms)**		
	One at time Jenkins	***DJB2***	***MOD***
2959	2.22	1.90	2.74
6175	4.41	4.27	4.73
9087	6.81	5.81	7.40

A tabela apresenta os resultados do desempenho das funções de *hash*, mediante o número de palavras que foram adicionadas. Percebe-se que a função *DJB2*, de Bernstein, obteve uma performance mais satisfatória em relação aos outros métodos. Isso ocorre devido aos chamados "números mágicos", que garantem uma alta velocidade de execução do programa e uma distribuição superior à *One at time Jenkins* e à MOD.

Gráfico I: Quantidade de *buckets* em relação ao tempo



O gráfico acima demonstra que o aumento do número de *buckets* viabilizou um aperfeiçoamento na performance do programa em razão de uma distribuição superior das chaves. No entanto, a ampliação constante da quantidade de *buckets*, ao longo da execução, é saturada, uma vez que esse número excedeu e causou um efeito direto no tempo de realização. O desempenho do código tende a estagnar a partir de valores maiores que 49783 na tabela.

A. *Função de hash em C e Python*

Tabela II: Comparação entre as funções em C e *Python*

Nº de palavras	**Tempos médio (ms)**	
	C (DJB2)	***Python***
2959	1.90	4.12
6175	4.27	7.37
9087	5.81	11.48

A tabela expõe os resultados da comparação entre os testes na função de espalhamento que obteve o melhor desempenho na linguagem C (*DJB2*) e em uma função de *hash* nativa da linguagem *Python*. A função em *Python* demonstrou menor eficiência, uma vez que não é viável utilizar a matriz associativa em uma base de dados grande. Em consequência disso, a *dict* retardou a execução das informações, enquanto a função de Bernstein se mostrou mais rápida, ratificando a qualidade dos chamados "números mágicos"na performance do código.

IV. Considerações finais

Diante do exposto, evidencia-se que não há grandes divergências entre os resultados obtidos em detrimento das funções. As análises quantitativas demonstram que houve uma melhoria de 14,54% da função *DJB2* em relação à Jenkins e 30,65% ao método da divisão. Ademais, observou-se uma saturação na execução com o aumento excessivo dos buckets, já que há um gasto superior de memória. Por fim, é possível visualizar que, em comparação com uma função de *hash* nativa da linguagem *Python*, a *DJB2* apresentou uma resposta mais rápida nos testes efetuados.

V. Trabalhos futuros

Como trabalhos futuros, pode-se apontar:

- A mudança no tratamento de colisões utilizando *linear probing* a fim de explorar aperfeiçoamento e comparação na distribuição de chaves;
- Utilização do *quadratic probing* para a otimização do código.

Referências

[1] CORMEN, T. H.; RIVEST, R. L.; LEISERSON, C. E.; STEIN, C.. Algoritmos: teoria e prática. Elsevier, 2012.

[2] Aaron M. Tenenbaum, Y. L. Estruturas de dados usando C. São Paulo: Makron Books, 1995.

[3] AUTODESK.HELP. To Use Spell Check, 2019. Disponível em:<https://knowledge.autodesk.com/support/inventor/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/Inventor-Help/files/GUID-2DEFA541-B829-4858-ADC1-DC4918F40E6F-htm.html >. Acesso em: 28 de agosto de 2019

[4] Jenkins. B. Dr. Dobbs Jornal, 1997. Disponível em: <https://www.burtleburtle.net/bob/hash/doobs.html>. Acesso em: 26 de agosto de 2019

[5] Mathur, Milind. Analysis of Parallel Lempel-Ziv Compression Using CUDA, 2013.

[6] Feofiloff, Paulo. Hashing, 2018. Disponível em: <https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/st-hash.html>. Acesso em: 26 de agosto de 2019

[7] Pantuza, Gustavo. Hashing, 2016. Disponível em: <https://blog.pantuza.com/artigos/tipos-abstratos-de-dados-tabela-hash>. Acesso em: 26 de agosto de 2019