UNIVERSIDADE FEDERAL DE MINAS GERAIS

DCC005 - ALGORITMOS E ESTRUTURAS DE DADOS III

HERNANE BRAGA PEREIRA

2014112627

TRABALHO PRÁTICO - 2

Belo Horizonte - MG
Junho/2018

INTRODUÇÃO

O objetivo deste trabalho é a resolução do problema da empresa *Doodle*, que oferece um serviço de busca de conteúdo na internet. Ela deseja implementar um algoritmo de recomendação de palavras para os usuários, onde dada uma palavra de busca, deve-se retornar as palavras de um dicionário que se adequam a um limite de mudanças máximo na palavra.

SOLUÇÃO DO PROBLEMA

Para solucionar o problema apresentado, foi demandado a escolha de pelo menos um paradigma de programação. O escolhido foi o paradigma de programação dinâmica.

A programação dinâmica consiste que dado um problema, ele possa ser modelado de forma a ser quebrado em partes menores, que possuem soluções ótimas. Calcula-se o ótimo de cada subestrutura e suas sobreposições geram uma solução final para o problema inicial. A programação dinâmica também permite estratégias de 'poda' para as soluções não desejadas.

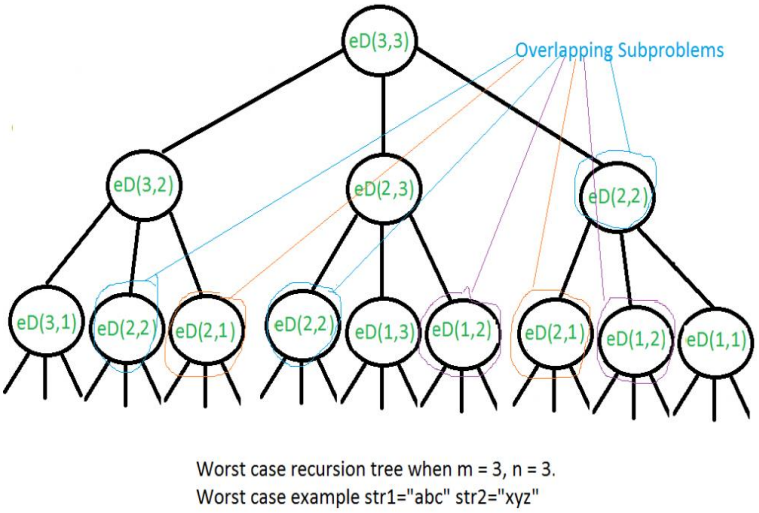

Figura 1. Exemplo de uma modelagem utilizando a abordagem de programação dinâmica

2.1 Modelagem do Problema

Para solucionar o problema proposto foi utilizado o algoritmo de programação dinâmica *Levenshtein Distance*, que consiste em calcular a menor distância de edição entre duas palavras, através de uma matriz (nm). Na matriz, ***n*** é o número de caracteres da palavra de origem e ***m*** o número de caracteres da palavra de destino. Este algoritmo é comumente utilizado em verificadores ortográficos. Abaixo o pseudocódigo que explica o funcionamento do algoritmo.

```
Função LevenshteinDistance(Caracter : str1[1..lenStr1], Caracter :
str2[1..lenStr2]) : INTEIRO
Início
   // tab é uma tabela com lenStr1+1 linhas e lenStr2+1 colunas
   Inteiro:  tab[0..lenStr1, 0..lenStr2]
   // X e Y são usados para iterar str1 e str2
   Inteiro:  X, Y, cost

   Para X de 0 até lenStr1
       tab[X, 0] ← X
   Para Y de 0 até lenStr2
       tab[0, Y] ← Y

   Para X de 1 até lenStr1
       Para Y de 1 até lenStr2
           Se str1[X] = str2[Y] Então cost ← 0
                                Se-Não cost ← 1 // Custo de substituição,
deleção e inserção é o mesmo neste trabalho, mas poderia ser diferente
           tab[X, Y] := menor(
                                tab[X-1, Y  ] + cost,     // Deletar
                                tab[X  , Y-1] + cost,     // Inserir
                                tab[X-1, Y-1] + cost      // Substituir
                            )
   LevenshteinDistance ← tab[lenStr1, lenStr2]
 Fim
```

Código 2. Pseudocódigo do algoritmo *Levenshtein Distance*

Edit_Dist	Nulo	M	A	R	C	A
Nulo	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2	2	1	2	3	3
R	3	3	2	1	2	3
T	4	4	3	2	2	3
A	5	5	4	3	3	**2**

Tabela 3. Exemplo de uma matriz criada pelo algoritmo, comparando as palavras “marca” e “carta”. Os pesos de edição, remoção e inserção são iguais. Distancia de edição é 2.

O pseudocódigo acima demonstra a forma mais simples de uso do algoritmo *Levenshtein Distance,* porém neste trabalho foi utilizada uma versão otimizada que utiliza O(*min*(m,n)) de espaço, ao invés de O(nm). Tal feito é realizado reutilizando o conteúdo previamente criado na matriz. Abaixo está o código em linguagem C da função *levenshtein,* presente no código fonte deste trabalho.

```c
/* Definicao que serve de suporte para o algoritmo Levenshtein Distance */
#define MIN3(a, b, c) ((a) < (b) ? ((a) < (c) ? (a) : (c)) : ((b) < (c) ? (b) : (c)))
/* Algoritmo Levenshtein para calcular distancia entre strings */
int levenshtein(char *s1, char *s2) {
    unsigned int s1len, s2len, x, y, lastdiag, olddiag;
    s1len = strlen(s1);  s2len = strlen(s2);
    unsigned int column[s1len+1];
    for (y = 1; y <= s1len; y++)
        column[y] = y;
    for (x = 1; x <= s2len; x++) {
        column[0] = x;
        for (y = 1, lastdiag = x-1; y <= s1len; y++) {
            olddiag = column[y];
            column[y] = MIN3(column[y] + 1, column[y-1] + 1, lastdiag + (s1[y-1] == s2[x-1] ? 0 : 1));
            lastdiag = olddiag;
        }}
    return(column[s1len]);
}
```

Código 2. Implementação em C do algoritmo *Levenshtein Distance* com otimização no uso de espaço. Link para o código original presente nas referências

Para armazenar e imprimir o resultado da busca, foi utilizada uma struct que armazena a palavra e seu valor de distância de edição associado. Após o resultado final ser computado, ele é ordenado utilizando a função *qsort* da biblioteca padrão do C e é impresso.

3.1 Análise Teórica do Custo Assintótico de Tempo

Podemos realizar a análise de custo assintótico avaliando o número de comparações que são realizadas pelo algoritmo. Como o algoritmo realiza (n + 1)*(m+1) comparações, seu custo assintótico de tempo é O(nm) para analisar cada palavra do dicionário. Portanto, o custo assintótico de tempo do projeto será: *tamanho_dicionário* * O(nm).

3.2 Análise Teórica do Custo Assintótico de Espaço

Para fazer a análise de custo de espaço, foca-se no tamanho da matriz que é criada para realizar os cálculos de distância. O algoritmo *Levenshtein Distance,* por padrão, ocupa espaço de O(nm), onde ***n*** e ***m*** são os caracteres que compõem as palavras que serão comparadas. Entretanto, neste trabalho foi utilizado uma variação do algoritmo que utiliza O(*min*(m,n)).

4.1 Testes do código

A máquina utilizada para rodar os testes possui 6,0 GB de memória RAM, com processador Core™ i7-5500U CPU 2.40Hz, com sistema operacional *Windows 10.*

ARQUIVO TESTE	TAMANHO DICIONARIO	PALAVRAS ENCONTRADAS	MÉDIA TEMPO (s)
input1.txt	15	5	0,2420
input2.txt	60	18	0,2410
input3.txt	90	27	0,2690
input4.txt	2000	600	0,7714
input5.txt	2000	600	1,2584
input6.txt	4000	1200	3,1474
input7.txt	6000	1800	6,0474
input8.txt	8000	2400	13,8084
input9.txt	10000	3000	23,2018
input10.txt	10000	3000	22,3630

Tabela 2: Nome dos arquivos de testes contendo: tamanho do dicionário, número de palavras encontradas e média de tempo em segundos gastos na execução do algoritmo

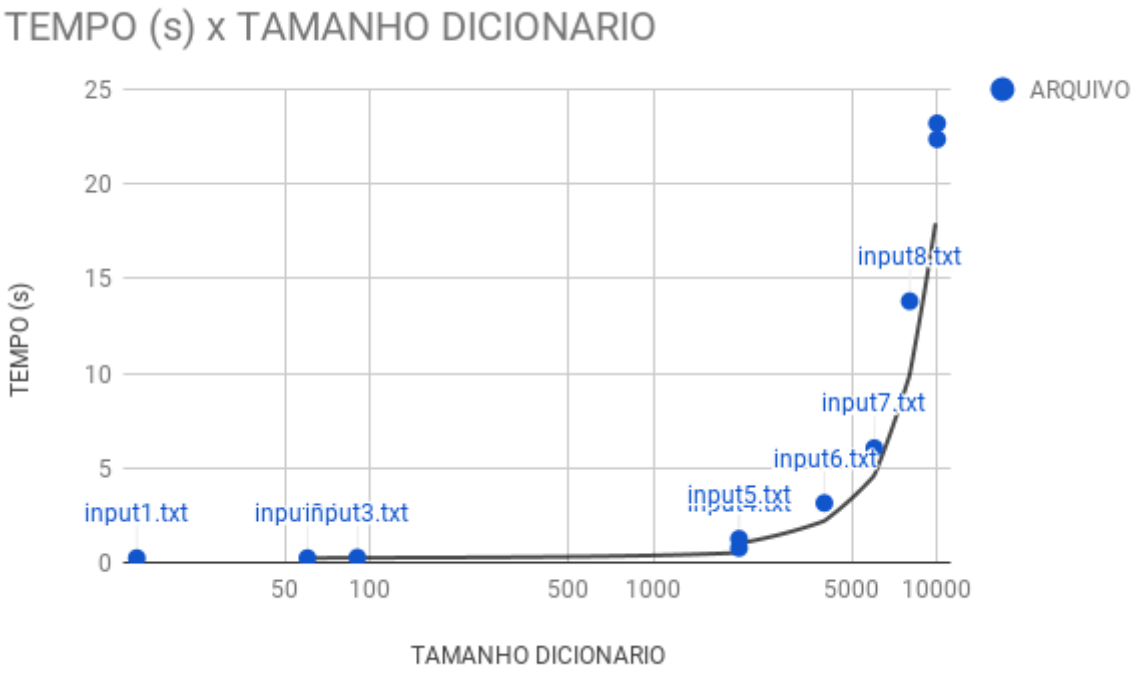

Gráfico 1: Tempo de execução x Tamanho de dicionário para cada arquivo de teste.

O gráfico demonstra que o tempo de execução do algoritmo é linear e varia com o tamanho do dicionário, de acordo com os casos de testes utilizados. Tal comportamento comprova a análise teórica do item 3 desta documentação.

5.1 Conclusão

Neste trabalho, foi resolvido o problema de recomendação de palavras da empresa *Doodle*. O problema foi resolvido utilizando o algoritmo de programação dinâmica *Levenshtein Distance*. A análise de complexidade teórica de tempo foi comprovada através de experimentos que utilizaram entradas grandes suficientes para analisar o comportamento assintótico.

REFERÊNCIAS

Geeks for Geeks, Uso de programação dinâmica no cálculo de distância de edição entre duas strings:
https://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/

Wikipédia, Artigo sobre o algoritmo *Levenshtein Distance*:
https://pt.wikipedia.org/wiki/Dist%C3%A2ncia_Levenshtein

Wikibooks, Implementação do algoritmo *Levenshtein Distance* de forma otimizada:
https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#C