Documentação - Trabalho Pratico 2 - AEDS III

Aluno – Jesimon Barreto Santos (2016070093)

1 Introdução

O Doodle é um serviço de busca de conteúdo pela internet com grande base de usuários. Você foi contratado pela Doodle para desenvolver um algoritmo que irá auxiliar seus usuários com problemas de digitação, o que é algo bastante comum entre os pessoas que digitam rapidamente. A ideia é implementar um sistema de recomendação de palavras parecidas com o que foi buscado, dado um limite de mudanças. Para isso, você recebe um conjunto de palavras D que serve como dicionário. Essas palavras fazem a indexação do conteúdo do Doodle. Além disso, você recebe uma palavra de consulta q e recebe um inteiro n, informando o máximo de operações para transformar uma palavra fonte em uma palavra destino. As seguintes operações são válidas e tem custo unitário:

1. **Inserção de caractere**: um caractere qualquer pode ser inserido na palavra de consulta.
2. **Remoção de caractere**: um caractere qualquer pode ser removido da palavra de consulta.
3. **Substituição**: um caractere qualquer da palavra de consulta pode ser alterado por outro caractere.

Sua função é retornar todas as palavras do dicionário que podem ser reconstruídas a partir da palavra de consulta, aplicando no máximo n operações. A sua saída deve mostrar as palavras que exigem menos alterações antes de palavras que exigem mais alterações. Caso 2 ou mais palavras exijam a mesma quantidade de alterações, você precisa desempatar usando a ordem léxica (i.e. alfabética).

2 Abordagem do Problema

Com base no problema apresentado, foi induzida uma abordagem baseado no algoritmo de Levenshtein Distance. Algoritmo usado para calcular a diferença mínima entre duas palavras de quaisquer dois tamanhos. Exatamente a quantidade de inserções, remoções ou substituição para transformar uma palavra na outra ou vice-versa.

Dada palavra1 e palavra2 de tamanhos respectivos s, t, respectivamente, o algoritmo usa uma matriz s + 1 por t + 1, minimizando o custo de adição a cada posição (i, j) da matriz representa o número de operações necessários para sair da palavra1 até a letra i até a palavra2 até a palavra j. Seguindo essa ideia, o exemplo dado na tabela 1 que representa a matriz resultado da comparação entre “dia” e “gia”. O número de operações necessárias é dado pela posição inferior direita. Nesse caso, é aplicado o paradigma Programação Dinâmica, pois, a formulação do problema (encontrar a quantidade de operações) é resolvido com a solução do subproblema (encontrar a quantidade de operações necessárias entre a primeira letra, depois das duas primeiras letras, e assim sucessivamente), com minimização entre as operações de custo associada a transformar a string na outra, até determinado ponto. A matriz é preenchida usando a formula:

- j, se i = 0;
- i, se j = 0;
- min(M(i-1,j) + 1, M(i, j- 1) + 1, M(i-1, j-1) + custo)
    - custo = 0, se os caracteres na posição referentes forem iguais;
    - custo = 1, se os caracteres forem diferentes.

Onde:

i é um número natural que vai de 0 até o tamanho da palavra1.

j é um número natural que vai de 0 até o tamanho da palavra2.

M é a matriz.

O número de operações entre "d" e "g" é 1 (na tabela, percebe-se posição [1,1]), diferença entre "di" e "gi", ainda se mantem 1 (na tabela, percebe-se posição [2,2] da matriz), seguindo essa ideia chegamos ao fim da palavra "dia" e "gia", que continua sendo 1.

Tabela 1. Resultado do algoritmo distância de Levenshtein para dia/gia.

		d	i	a
	0	1	2	3
g	1	**1**	2	3
i	2	2	**1**	2
a	3	3	2	**1**

Agora, o programa geral para resolver esse problema é dado por:

```
Read(D, n, q)

List[n]

For i in (0, D):

        Read(d)

        IF( LevenshteinDistance(q, d) < n)  List.add(d)

Sort(List)
```

3 Complexidade

A complexidade do programa é explicada em termos do algoritmo, pois é a principal implementação desse trabalho prático.

Iniciando com a explicação da complexidade de tempo, começando pelo algoritmo de Levenshtein Distance, como explicado anteriormente, é necessário preencher a matriz toda antes de finalizar, dessa maneira:

$$M*N$$

Onde:

M é número de caracteres da palavra modelo

N é o número de caracteres da palavra teste

Porém, a matriz é preenchida D vezes, onde D é o número de palavras de teste, logo, a parte do algoritmo de Levenshtein Distance tem complexidade de tempo dada por:

$$O (M*N*D)$$

A segunda parte é de ordenação, que usamos a função qsort disponível no bloco de biblioteca da linguagem C, dessa maneira, a ordenação é:

$$O (s*\log s)$$

Onde:

s é o número de palavras que tem menos de **n** operações para se tornar a palavra modelo.

Finalizada a complexidade de tempo, agora será explicada a complexidade de espaço, que é parecido com o cálculo anterior feito, pois a cada comparação entre duas palavras, é necessário salvar uma matriz de (m+1) X (n+1), além disso, um vetor de tamanho D para salvar o resultado de cada palavra.

$$E ([\{P+1\} * \{Q+1\}] + D)$$

Onde:

P é o número de letras na palavra modelo, ou, **q**.

Q é o número de letras na palavra de comparação, depende do **d.**

4 Experimentos

Os testes foram feitos em um computador com processador i5, 2.6 GHz e memória de 1 GB, no sistema operacional Ubuntu. Os casos de teste disponibilizados foram executados com sucesso, seguem o comportamento expresso como a seguir:

Tabela2. Tempo em segundos necessário para executar cada arquivo de teste.

Arquivo	tempo (s)
input1	0.00
input2	0.00
input3	0.00
input4	0.15
input5	3.80
input6	5.80
input7	7.10
input8	53.05
input9	93.46
input10	73.54

Como foi mostrado no tópico de Abordagem do problema, o algoritmo geral mostra que são ordenados apenas as palavras que tem até n operações. Fazendo uma análise comparativa diretamente com os casos input9 e do input10, os dois tem 10000 (valor de D) casos de comparação, porém diferenciam no n (input9 é 3, input10 é 4). Com base nisso, pode-se justificar o input9 demorar mais tempo pois existem mais palavras d com mudanças menores que o n dado, isso em comparação com o input10.

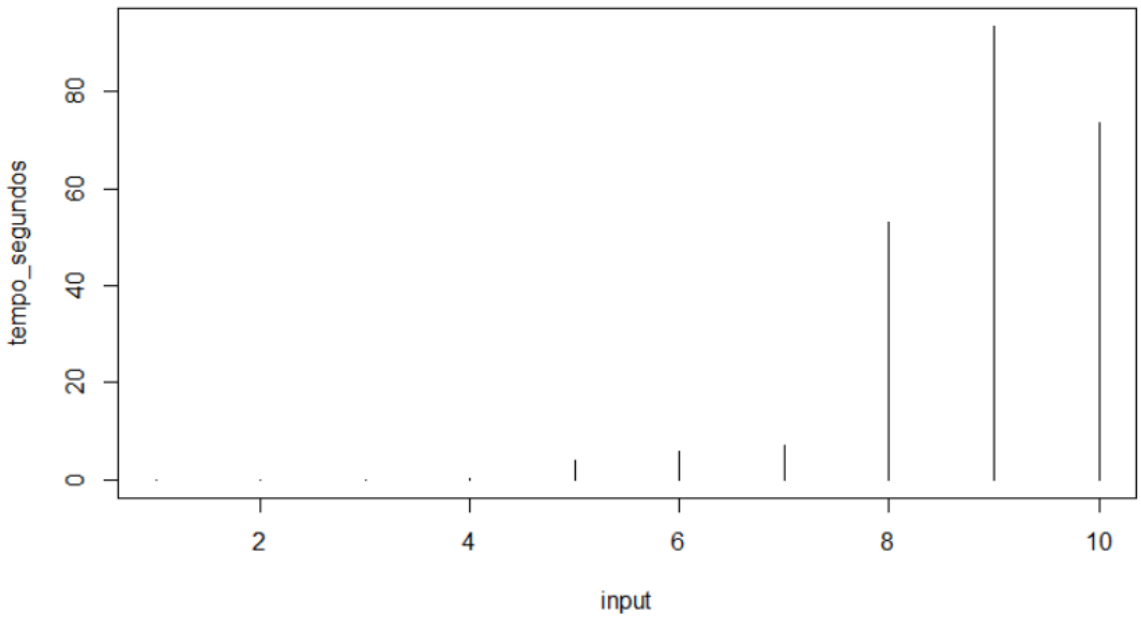

Figura1. Gráfico que relaciona tempo em segundos com os inputs de teste.

O gráfico exibe, em formato de histograma, o resultado de tempo por entrada. O eixo x é o input e o y é o tempo em segundos da execução de cada entrada.

5 Conclusão

Esse programa foi criado para resolver um problema de contagem de distância entre várias palavras, cujo, cada palavra era comparada com uma de modelo, dessa maneira, contabilizar a quantidade de operações mínimas necessárias para transformar a palavra de testa na palavra modelo, por último deve exibir as palavras, que tem número de operações menor ou igual a um dado limite ‘n’, em ordem crescente de operações e caso de igualdade em ordem alfabética.

6 Referencial Teórico

Ziviani, N. *Projeto de Algoritmos com Implementações em Pascal e C*, São Paulo, Brazil, Cengage Learning, ISBN 13 978-85-221-1050-6, 2010, third edition reviewed and extended, 659 pages (in Portuguese).

André Backes. *Linguagem C: Completa e Descomplicada*. Editora Campus Elsevier, 2012.

Michael Gilleland, Merriam Park Software**.** *Levenshtein Distance*. Acesso em 29 de maio de 2018;https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

http://www.levenshtein.net/ . Acesso em 29 de maio de 2018.