ENGENHARIA DE SOFTWARE

Desafio: Análise de Similaridade Textual

Disciplina: Dynamic Programming

Professor: Lucas Mendes Marques Gonçalves

Integrantes - RM

Arnaldo de Moraes Pereira filho – 555780

Augusto Barcelos Barros – 565065

Erick Moreira Fujita – 556096

FIAP

Sumário

1. Explicação das distâncias:

S1 = Arnaldo

S2 = Augusto

(Observação: Nos exemplos tratarei todos os algoritmos como case-insensitive)

a) Longest Common Subsequence (LCS)

Definição: Algoritmo que encontra a **maior sequência de caracteres** que aparece na mesma ordem entre S1 e S2, porém não precisa ser contíguo (pode haver caracteres entre eles)

Análise: O “a” inicial é correspondente (Comprimento + 1 = 1)
“u” de “augusto” não em existe em “rnaldo”, “g” não existe, “u” não existe, “s” não existe, “t” não existe
“o” existe no final de “Arnaldo” (Comprimento + 1 = 2)

Resultado: A subsequência mais longa é **2**, que no caso seria **“ao”**.

Tabela Dinâmica:

	•	A	U	G	U	S	T	O
•	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
R	0	1	1	1	1	1	1	1
N	0	1	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1
L	0	1	1	1	1	1	1	1
D	0	1	1	1	1	1	1	1
O	0	1	1	1	1	1	1	**2**

$$L[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ 1 + L[i-1][j-1] & \text{se } s1[i] = s2[j] \\ \max(L[i-1][j], L[i][j-1]) & \text{se } s1[i] \neq s2[j] \end{cases}$$

b) Longest Common Substring

Definição: Algoritmo que encontra a **maior sequência de caracteres contíguos** (um seguido do outro) que existe entre S1 e S2.

Análise: O “a” inicial é comum (Comprimento 1), depois disso “u” é diferente de “r”, então a sequência contígua é quebrada, e apenas no final “o” será comum (comprimento 1).

Resultado: O comprimento máximo é **1**, que no caso seria **“a”** e **“o”**

Tabela Dinâmica:

	•	A	U	G	U	S	T	O
•	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0
R	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0
O	0	0	0	0	0	0	0	1

$$S[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ 1 + S[i-1][j-1] & \text{se } s1[i] = s2[j] \\ 0 & \text{se } s1[i] \neq s2[j] \end{cases}$$

c) Edit Distance – Levenshtein

Definição: Algoritmo que calcula o número **mínimo** de operações de edição (entre inserção, remoção e substituição) para transformar S1 em S2, ou seja, o quão diferentes as duas strings são.

Análise: Nesse caso o menor caminho possível seria fazendo 5 substituições, ou seja, **distância 5**

```
“a” > “a” (Custo 0 - corresponde)
“u” > “r” (Custo 1 - substituição)
“g” > “n” (Custo 1 - substituição)
“u” > “a” (Custo 1 - substituição)
“s” > “l” (Custo 1 - substituição)
“t” > “d” (Custo 1 - substituição)
“o” > “o” (Custo 0 - corresponde)
```

Tabela Dinâmica:

	•	**A**	**U**	**G**	**U**	**S**	**T**	**O**
•	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
R	2	1	1	2	3	4	5	6
N	3	2	2	2	3	4	5	6
A	4	3	3	3	3	4	5	6
L	5	4	4	4	4	4	5	6
D	6	5	5	5	5	5	5	6
O	7	6	6	6	6	6	6	**5**

$$D[i][j] = \begin{cases} j & \text{se } i = 0 \text{ (Inserções)} \\ i & \text{se } j = 0 \text{ (Remoções)} \\ D[i-1][j-1] & \text{se } s1[i] = s2[j] \text{ (Match)} \\ 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1]) & \text{se } s1[i] \neq s2[j] \text{ (Remoção, Inserção, Substituição)} \end{cases}$$

Onde:

- `D[i-1][j]` representa uma **remoção**.
- `D[i][j-1]` representa uma **inserção**.
- `D[i-1][j-1]` representa uma **substituição**.

2. Implementação do Algoritmo de Levenshtein

Bottom Up

```python
def levenshtein_distance_iter(s1, s2):
    """
    Calcula a distância de Levenshtein entre duas strings de forma iterativa (bottom-up).

    Complexidade de tempo: O(m * n)
    """
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if s1[i - 1].lower() == s2[j - 1].lower() else 1
            dp[i][j] = min(
                dp[i - 1][j] + 1,  # remoção
                dp[i][j - 1] + 1,  # inserção
                dp[i - 1][j - 1] + cost,  # substituição
            )
    return dp[m][n]
```

Top Down

```python
def levenshtein_distance_rec(s1, s2):
    """
    Calcula a distância de Levenshtein entre duas strings usando recursão e cache manual.

    Complexidade de tempo: O(m * n)
    Complexidade de tempo sem cache: O(3^(m+n))
    """
    cache = {}

    def rec(i, j):
        if (i, j) in cache:
            return cache[(i, j)]
        if i == 0:
            result = j
        elif j == 0:
            result = i
        else:
            cost = 0 if s1[i - 1].lower() == s2[j - 1].lower() else 1
            result = min(
                rec(i - 1, j) + 1,  # remoção
                rec(i, j - 1) + 1,  # inserção
                rec(i - 1, j - 1) + cost,  # substituição
            )
        cache[(i, j)] = result
        return result

    return rec(len(s1), len(s2))
```

Comparação

Teste Básico com 'Augusto' e 'Augto'

Método	Distância	Tempo (ms)
Iterativo (Bottom-Up)	2	0,02
Recursivo (Top-Down)	2	0,04

Resultados iguais: True

Teste de Desempenho com 100 caracteres

Método	Distância	Tempo (ms)
Iterativo (Bottom-Up)	93	2,66
Recursivo (Top-Down)	93	11,17

Resultados iguais: True

3. Análise LLMS

Prompts utilizados:

Fatuais

- p1. Qual o rio mais longo do mundo?
- p2. Quem foi a primeira pessoa a pisar na Lua?

Criativos

- p3. Escreva um poema curto (quatro linhas) sobre uma xícara de café.
- p4. Descreva uma cidade futurista em um único parágrafo.

Código

- p5. Escreva uma função em Python que inverte uma *string*.
- p6. Escreva a estrutura básica de um arquivo HTML5 (apenas as tags *head* e *body*).

Explicações Simples

- p7. O que é um '*hash map*' (ou dicionário) em programação? Explique de forma simples.
- p8. Explique o que é a fotossíntese para uma criança de 10 anos.

Opiniões

- p9. Qual é melhor: iOS ou Android? Liste uma vantagem de cada.
- p10. O trabalho remoto é o futuro do trabalho? Justifique brevemente.

Distâncias de Levenshtein (Primeiros 100 chars)

Prompt	ChatGPT vs DeepSeek	ChatGPT vs Gemini	DeepSeek vs Gemini
P1	78	73	76
P2	59	77	35
P3	78	79	82
P4	78	80	82
P5	49	70	72
P6	0	0	0
P7	56	71	66
P8	56	61	66
P9	80	88	68
P10	81	77	73

Distância Geral (Média & Mediana)

Comparação	Média Da Distância	Mediana da Distância
ChatGPT vs DeepSeek	61.50	68.50
ChatGPT vs Gemini	67.60	75.00
DeepSeek vs Gemini	62.00	70.00

4. O Grande

```
Gerando uma lista REVERSA de 1000 elementos (pior caso para o Insertion Sort)...

Executando Insertion Sort (O(n²))...
Tempo de execução: 49,50 ms

Executando Merge Sort (O(n log n))...
Tempo de execução: 1,52 ms
```

Análise de Complexidade (Big O)

Algoritmo	Complexidade (Pior Caso)
Insertion Sort	O(n²)
Merge Sort	O(n log n)

Análise de Consumo e Proporção (n=1000)

Qual consome mais energia?
O consumo de energia é proporcional ao tempo de CPU.
O **Insertion Sort** levou **49,50 ms**, enquanto o **Merge Sort** levou **1,52 ms**.
Claramente, o **Insertion Sort consumiu muito mais energia.**

Qual a proporção?
A proporção teórica de operações é $(n^2 / n \log n) = (n / \log n)$.
Com n=1000 e $\log_2(1000) \approx 9.97$, a proporção teórica é:
1000 / 9.97 ≈ 100.34 vezes

A proporção PRÁTICA (medida pelo tempo) foi:
49,50 ms / 1,52 ms ≈ 32.57 vezes