UFRPE/UAG
Projeto e Análise de Algoritmos, Turma 2018.2
Professor: Tiago Buarque Assunção
Alunos: Mateus Baltazar de Almeida e Matheus Machado Vieira

Relatório do projeto: *Distância de Edição*

Garanhuns - PE
21/01/2019

1. Entendendo o problema

O problema da *Distância de Edição* tem como intuito transformar uma string A em uma string B. Melhor explicando; precisamos transformar a cadeira de carácter *x*[1..m] em uma nova cadeia *y*[1..n], havendo a possibilidade de executar vários tipos de operações de transformação cujo possuem custos individuais ao serem realizadas. Dessa forma, o objetivo é encontrar o custo do melhor conjunto de operações de transformação que mudam *x* para *y*. A cadeia *z* representará o resultado final, ou seja, para *j = 1, 2, …, n*, teremos que *z*[j] = *y*[j]. Sendo assim, iremos fazer mudanças apenas na cadeia *z*.

No nosso caso, teremos 5 tipos de transformação (sendo *i* o representante dos índices de *x* e j o representante dos índices de *z*). São estas as transformações:

- Copiar: copia um carácter de *x* para *z*, definido como: *z*[*j*] = *x*[*i*];
  i e *j* são incrementados em 1.
- Substituir: substitui o caracter *x*[i] por um outro carácter *c* em *z*[j], definido como: *z*[j] = *c;*
  i e *j* são incrementados em 1.
- Deletar: ignora um carácter de *x* incrementando o *i*. E permanecendo com o *j* inalterado.
- Inserir: incrementa o carácter *c* em *z*[*j*];
  j é incrementado em 1, e *i* permanece inalterado.
- Trocar: os dois caracteres seguintes são copiados na ordem inversa de *x* para *z*. Ou seja: *z*[*j*] = *x*[*i*+1] e *z*[*j*+1] = *x*[*i*];
  i e *j* são incrementados em 2.

2. Entendendo como resolver o problema

Utilizando a forma de resolução baseada no método *Damerau–Levenshtein distance*, na qual envolve os 5 casos de transformação explicados. Desenvolvemos um algoritmo que trabalha utilizando uma matriz bidimensional na qual possui *m+1* linhas e *n+1* colunas, sendo respectivamente *m* e *n* a quantidade de caracteres que compõem as string *x* e *y*. Nessa matriz, a primeira linha e primeira coluna são enumeradas de 0 até *m* (primeira linha) e de 0 até *n* (primeira coluna), onde tanto a linha, quanto a coluna estão representando os caracteres das demais strings no intervalo [1..m] e [1..n].

Para auxílio ao entendimento do funcionamento da forma desenvolvida, utilizaremos como exemplo a string *x* = “cytab” e a string *y* = “cyukba”. Ou seja, queremos transformar *x* em *y* aplicando as operações vistas, de forma que o conjunto de ações realizadas seja o menos custoso. Atribuindo custos às operações:

copiar = 0; substituir = 2; inserir = deletar = 1; trocar = 1;

Assim, podemos dar continuidade ao entendimento da resolução.

Iniciando a matriz e atribuindo os valores da primeira linha e primeira coluna, temos:

	y→	c	y	u	k	b	a
x↓	0	1	2	3	4	5	6
c	1						
y	2						
t	3						
a	4						
b	5						

Feito isso, o algoritmo percorrerá para cada caracter da string *x* (índice *i*), todos os outros caracteres da string *y* (índice *j*), e a cada iteração dessa, será atribuído um valor à matriz na posição (*i*, *j*) que corresponde ao valor mínimo destas 5 operações (a melhor operação para aquele subproblema):

```
if(x[i] == y[j]) {
        matriz[i, j] = custo_copiar
} else {
        matriz[i, j] =   min(   matriz[i-1, j] + custo_remover,
                                matriz[i, j-1] + custo_inserir,
                                matriz[i-1, j-1] + custo_substituir
                        )
        if(j>1 && i>1 && x[i] == y[j+1] && x[i+1] = y[j]) {
                matrix[i, j] =   min(   matriz[i, j],
                                        matriz[i-2, j-2] + custo_trocar
                                )
        }
}
```

Dessa maneira, as primeiras verificações para quando *i* = 1 atribuiriam à tabela os seguintes valores:

		c	y	u	k	b	a
	0	1	2	3	4	5	6
c	1	0	1	2	3	4	5
y	2						
t	3						
a	4						
b	5						

Seguindo o algoritmo e completando os demais itens da tabela, teremos:

		c	y	u	k	b	a
	0	1	2	3	4	5	6
c	1	0	1	2	3	4	5
y	2	1	0	1	2	3	4
t	3	2	1	2	3	4	5
a	4	3	2	3	4	5	4
b	5	4	3	4	5	4	4

O nosso resultado sempre será o valor contido na célula do canto inferior direito da matriz, que corresponde ao somatório do custo de todas as transformações (custo do melhor conjunto de transformações).

Para “trilharmos o caminho percorrido”, basta seguirmos os t passos:

Após o término da execução de cálculo de custo, começando a observar da célula que contém o resultado final;

Ao observar uma célula, faça:

- Marque a célula atual (pinte-a)
- Se a transformação realizada para chegar na célula atual foi de
    - cópia, observe a célula: 1 para cima, 1 a esquerda;
    - inserção, observe a célula: 1 para esquerda
    - remoção, observe a célula: 1 para a cima
    - substituição, observe a célula: 1 para cima, 1 a esquerda;
    - troca, observe a célula: 2 para cima, 2 a esquerda;

Dessa forma, teremos:

		c	y	u	k	b	a
	0	1	2	3	4	5	6
c	1	0	1	2	3	4	5
y	2	1	0	1	2	3	4
t	3	2	1	2	3	4	5
a	4	3	2	3	4	5	4
b	5	4	3	4	5	4	4

3. Implementação

Implementamos de três maneiras diferentes: uma forma iterativa *bottom up*, recursiva com memorização e recursiva sem memorização.

(Quanto a implementação, recomenda-se observar o código, pois por lá está melhor de entender. Está bem documentado)

4. Resultado

4.1. Sempre o melhor caso

Sempre obteremos o melhor caso (menor custo de transformação) pelo fato de que todo resultado de cada passo será o mínimo entre as possíveis transformações. Desse modo, para cada passo (subproblema), obteremos a melhor transformação de acordo com seu custo. E como sabemos, uma resposta composta de resoluções ótimas de subproblemas, forma um resultado ótimo. Logo, para qualquer caso, teremos o melhor caso.

4.2. Custo

Uma coisa que é válida notar ao observar o tempo de processamento de cada uma das abordagens implementadas é a diferença entre a forma recursiva sem memorização e as demais. Nos casos a seguir, enquanto as formas iterativa e recursiva com memorização resultam quase que instantâneamente, observemos as strings e o tempo para ser calculado cada caso usando a forma recursiva sem memorização (é válido observar a diferença entre as strings de cada caso e relacionar com a diferença de tempo):

str1 = "atraocaasdq"; str2 = "rtoacabdwqo";
Tempo: 0m1,299s

str1 = "atraocaasdqw"; str2 = "rtoacabdwqoi";
Tempo: 0m5,763s

str1 = "atraocaasdqwa"; str2 = "rtoacabdwqoid";
Tempo: 0m27,834s

str1 = "atraocaasdqwad"; str2 = "rtoacabdwqoids";
Tempo: 1m46,149s

str1 = "atraocaasdqwadm"; str2 = "rtoacabdwqoids";
Tempo: 2m44,929s

5. Dificuldades do projeto

A maior dificuldade foi acharmos exemplos para compararmos os resultados obtidos da nossa implementação, além disso, notamos que não é tão fácil achar material relacionado a esse caso específico do problema por conta da sua popularidade, comparada a por exemplo o problema do caixeiro-viajante, mas de qualquer forma, achamos.

Ademais, não obtivemos maiores problemas.