UNIVERSIDAD
DE GRANADA

Programación Dinámica

Algorítmica. Práctica 4

Jose Alberto Hoces Castro
Javier Gómez López
Manuel Moya Martín Castaño

Mayo 2022

Contenidos

Objetivo de la práctica

El objetivo de esta práctica es aprender a implementar y utilizar algoritmos que utilizan la técnica de programación dinámica. Para ello hemos tenido que resolver el siguiente ejercicio:

Introducción

Enunciado

Dos hermanos fueron separados al nacer y mediante un programa de televisión se han enterado que podrían ser hermanos. Ante esto, los dos están de acuerdo en hacerse un test de ADN para verificar si realmente son hermanos.

Enunciado

Dadas las 2 entradas:

PRIMERA

Hermano 1 - abbcdefabcdxzyccd

Hermano 2 - abbcdeafbcdzxyccd

SEGUNDA

Hermano 1 - 0101110001000101010100100010010010001

Hermano 2 - 1100001001001010100010100100110101000

- Deben encontrar el % de similitud que existe entre estos posibles hermanos, como es un ejemplo lo haremos para 2 entradas posibles.
- Dar las salidas (secuencia más larga) de las 2 entradas.
- Dar la matriz de los cálculos de la primera entrada.

Requisitos de la programación dinámica

Requisitos de la Programación Dinámica

- **Naturaleza n-etápica**
- **Verificación del POB (Principio de Optimalidad de Bellman)**
- **Planteamiento de una recurrencia**

Requisitos de la Programación Dinámica. Recurrencia planteada

1. Comenzaremos rellenando la primera fila y la primera columna de ceros.

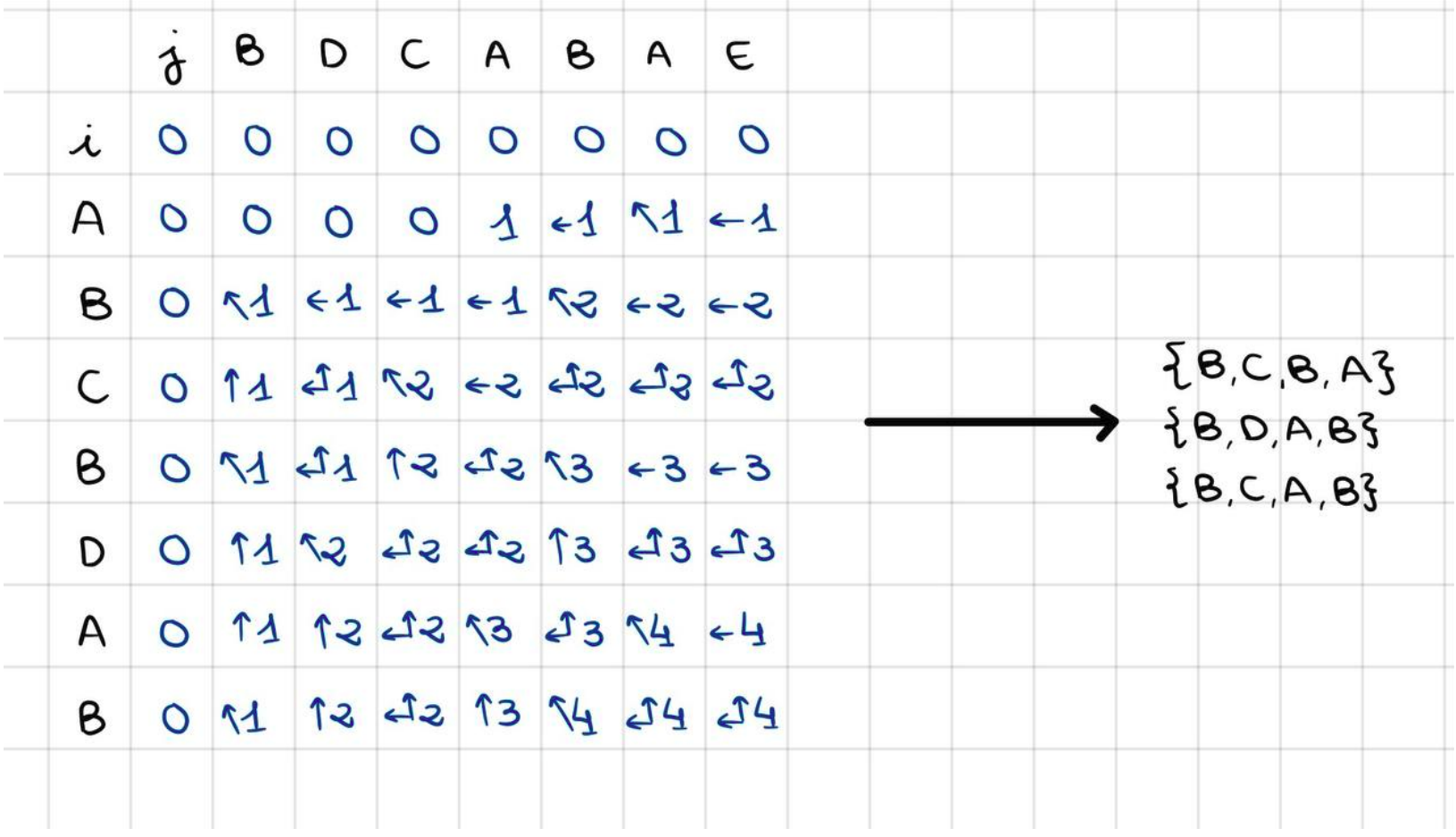
2. Rellenamos el resto de la matriz según el siguiente criterio:

$$A(i,j) = \begin{cases} 0 & si \quad i = j = 0 \\ max(A(i-1,j), A(i,j-1)) & si \quad x_i \neq y_j \\ A(i-1,j-1) + 1 & si \quad x_i = y_j \end{cases}$$

3. Por último, hallamos el máximo de la matriz (que estará situado en la última fila) deshaciendo el camino que hemos ido haciendo por las casillas y teniendo en cuenta si estamos en una posición de coincidencia o no.

Requisitos de la Programación Dinámica. Ejemplo

Dado un ejemplo pequeño de secuencias, sean < A, B, C, B, D, A, B > y < B, D, C, A, B, A, E >, la matriz usando nuestro algoritmo quedaría así:



Requisitos de la PD. Cálculo de una solución

1. Cálculo de la matriz

```
for(int i = 1; i < n+1; i++){
    for(int j = 1; j < m+1; j++){
        if(entrada1[i-1] == entrada2[j-1]){
            matriz_calculos[i][j] = matriz_calculos[i-1][j-1]+1;
        }
        else{
            if(matriz_calculos[i][j-1] >= matriz_calculos[i-1][j
    ]){
                matriz_calculos[i][j] = matriz_calculos[i][j-1];
            }
            else{
                matriz_calculos[i][j] = matriz_calculos[i-1][j];
            }
        }
    }
}
```

Requisitos de la PD. Cálculo de una solución

2. Localización del máximo y obtención de la subsecuencia más larga

```
1  int max = 0;
2  int posj = 0;
3
4  for(int j = 1; j < m+1; j++){
5      if(matriz_calculos[n][j] >= max){
6          max = matriz_calculos[n][j];
7          posj = j;
8      }
9  }
10
11 list<char> resultado;
12 int i = n;
13
```

Requisitos de la PD. Cálculo de una solución

2. Localización del máximo y obtención de la subsecuencia más larga

```
while(i >= 1){
    if(entrada1[i-1] == entrada2[posj-1]){
        resultado.push_front(entrada2[posj-1]);
        posj--;
        i--;
    }
    else{
        if(matriz_calculos[i][posj-1] == matriz_calculos[i][posj
    ]){
            posj--;
        }
        else{
            i--;
        }
    }
}
```

Solución

Solución. Secuencias más largas

SECUENCIA MÁS LARGA LETRAS:

abbcdeabcdzyccd

SECUENCIA MÁS LARGA NÚMEROS:

11000000101010100100010010010O

Solución. Similitud

PORCENTAJE DE SIMILITUD LETRAS: 88.2353 %

PORCENTAJE DE SIMILITUD NÚMEROS: 83.3333 %

Solución. Matriz de cálculos del primer problema

		a	*b*	*b*	*c*	*d*	*e*	*a*	*f*	*b*	*c*	*d*	*z*	*x*	*y*	*c*	*c*	*d*
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
b	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
b	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
c	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4
d	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5
e	0	1	2	3	4	5	6	6	6	6	6	6	6	6	6	6	6	6
f	0	1	2	3	4	5	6	6	7	7	7	7	7	7	7	7	7	7
a	0	1	2	3	4	5	6	7	7	7	7	7	7	7	7	7	7	7
b	0	1	2	3	4	5	6	7	7	8	8	8	8	8	8	8	8	8
c	0	1	2	3	4	5	6	7	7	8	9	9	9	9	9	9	9	9
d	0	1	2	3	4	5	6	7	7	8	9	10	10	10	10	10	10	10
x	0	1	2	3	4	5	6	7	7	8	9	10	10	11	11	11	11	11
z	0	1	2	3	4	5	6	7	7	8	9	10	11	11	11	11	11	11
y	0	1	2	3	4	5	6	7	7	8	9	10	11	11	12	12	12	12
c	0	1	2	3	4	5	6	7	7	8	9	10	11	11	12	13	13	13
c	0	1	2	3	4	5	6	7	7	8	9	10	11	11	12	13	14	14
d	0	1	2	3	4	5	6	7	7	8	9	10	11	11	12	13	14	15

Conclusiones

Conclusiones

- Si hubiésemos resuelto este problema usando la **fuerza bruta**, es decir, enumerando todas las subsecuencias comunes existentes, obtendríamos un algoritmo de orden $O(d^n)$, siendo d la longitud de las cadenas.
- Gracias a la **programación dinámica**, evitamos muchos de los cálculos que se harían en **fuerza bruta**, obteniendo así un algoritmo de **orden polinomial**, en este caso concretamente $O(n \cdot m)$, siendo n y m las longitudes de las secuencias de entrada.

Conclusiones

- Obtenemos la **solución optimal**, cosa que no habríamos logrado con otras técnicas ya vistas como **Greedy**
- También ganamos ventaja respecto a **Divide y Vencerás**, pues en **programación dinámica** tenemos problemas que están **encajados**, por lo que **no repetimos cálculos**.
- Como principal **inconveniente** de la **programación dinámica**, se hace un **gran uso de recursos**, en nuestro caso, la **memoria** reservada para nuestra **matriz de cálculos**.