Algorítmica
Programación Dinámica

Conrado Martínez
U. Politècnica Catalunya

ALG Q2-2021–2022

Temario

Parte IV

Programación Dinámica

Parte IV

Programación Dinámica

- Introducción a la Programación Dinámica
- Mochila Entera
- Distancia de edición y otros problemas sobre strings
- Caminos Mínimos en Grafos

Introducción

La programación dinámica (PD) es un esquema de resolución de problemas de optimización que se fundamenta en dos ingredientes fundamentales:

1. El principio de optimalidad (Bellman & Dreyfus, 1962)

   "Un problema satisface el principio de optimalidad si cualquier subsolución (solución parcial) de la solución óptima de una instancia es solución óptima de la correspondiente subinstancia."

2. La memoización: las soluciones de los subproblemas se almacenan en una tabla para evitar hacer más de una vez la misma llamada recursiva; de hecho, con la memoización se elimina la recursividad y se concluye con un algoritmo iterativo

Son muchos los problemas que admiten una solución mediante PD. El esquema es conceptualmente recursivo, pero por lo general se trabaja con versiones iterativas, organizando cuidadosamente el orden de resolución de los subproblemas y utilizando la *memoización*, es decir, almacenando resultados intermedios en memoria con el fin de optimizar su rendimiento.

Algunos problemas que se resuelven mediante este esquema incluyen:

- Cálculo de la distancia de edición entre cadenas.
- Cálculo de los caminos mínimos entre todos los pares de vértices de un grafo (algoritmo de Floyd).
- Cálculo del BST ponderado estático óptimo.
- El problema de la mochila entera.
- El problema del viajante de comercio.
- Cálculo de la derivación incontextual más verosímil.
- Cálculo de los caminos mínimos desde un vértice a los restantes, con pesos eventualmente negativos (algoritmo de Bellman-Ford).
- Cálculo de la secuencia de mezclas óptima.
- Cálculo de la clausura transitiva de un grafo (algoritmo de Warshall).
- Cálculo de la cadena de multiplicación de matrices óptima.

Pasos para desarrollar una solución de programación dinámica:

1. Establecer una recurrencia para el valor (coste, beneficio) de una solución óptima de la instancia dada en términos de los valores a soluciones óptimas de subinstancia, aplicando el principio de optimalidad
2. Diseñar la estructura de datos apropiada para almacenar soluciones intermedias (memoización) y determinar el orden en que dichas soluciones intermedias deben obtenerse (cómo “rellenar” la tabla).

3. Implementar el algoritmo iterativo, calculando su coste en tiempo y espacio
4. Detectar posibles optimizaciones en tiempo y/o espacio
5. Diseñar el algoritmo que reconstruye una solución óptima, utilizando la tabla de memoización o alguna estructura de datos auxiliar que permite “recordar” cuáles son las subinstancias que proporcionan las soluciones parciales con las que se construye la solución óptima.

Parte IV

Programación Dinámica

- Introducción a la Programación Dinámica
- Mochila Entera
- Distancia de edición y otros problemas sobre strings
- Caminos Mínimos en Grafos

Mochila Entera

Mochila Entera (0-1 KNAPSACK): Dado un conjunto de n objetos donde cada objeto tiene un peso $w_i \in \mathbb{Z}^+$ (los pesos son todos enteros positivos)y un valor v_i, y dado peso máximo permitiod en la mochila $W \in Z^+$, el objetivo es hallar un subconjunto $A \subseteq \{1, \ldots, n\}$ que maximiza el valor sin exceder el peso, esto es:

1. El valor

$$\sum_{i \in A} v_i$$

   es el máximo entre todos los subconjuntos A posibles, esto es, aquellos que satisfacen
2. $\sum_{i \in A} w_i \leq W$

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n-1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de ítems posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n-1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de ítems posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n - 1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de items posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n-1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de items posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n - 1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de items posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Entrada: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Sea $S \subseteq \{1, \ldots, n\}$ una solución óptima del problema, siendo el beneficio máximo conseguido $\sum_{i \in S} v_i$
- Consideremos el último elemento; tenemos dos casos posibles:
  - $n \notin S$, entonces S es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, entonces $S' = S \setminus \{n\}$ es una solución óptima para el problema $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- Podemos aplicar el principio de optimalidad; en ambos casos, obtenemos la solución óptima buscada considerando solo los elementos del 1 al $n - 1$ y con capacidad máxima de la mochila igual a W or igual a $W - w_n$
- Consideraremos subproblemas de la forma (i, x) representando que el conjunto de items posibles es $\{1, \ldots, i\}$ y que la capacidad máxima de la mochila es x.

Mochila Entera

Sea $V_{i,x}$ el beneficio máximo conseguible con un subconjunto de los elementos 1 a i y capacidad máxima x.

1. Nos interesa $V_{n,W}$, ese el beneficio máximo obtenible en el problema original
2. Casos de base: (1) para todo x, $V_{0,x} = 0$: sin objetos de dónde elegir no hay beneficio; (2) para todo i y todo $x \leq 0$, $V_{i,x} = 0$: sin capacidad en la mochila no hay beneficio
3. Caso recursivo: si $x \geq 0$ y $i \geq 1$ entonces

$$V_{i,x} = \text{máx}\{V_{i-1,x-w_i} + v_i, V_{i-1,x}\}$$

Mochila Entera: Algoritmo de PD

Sea V una tabla $(n+1) \times (W+1)$ de manera que $V[i,x]$ guardará $v_{i,x}$. Para rellenar dicha tabla debemos proceder de arriba a abajo y de izquierda a derecha: para calcular $V[i,x]$ se necesitan los elementos $V[i-1,y]$ de la fila previa.

```
procedure KNAPSACK(i, x)
    for i := 0 to n do
        V[i, 0] := 0
    end for
    for x := 1 to W do
        V[0, x] := 0
    end for
    for i := 1 to n do
        for x := 1 to W do
            V[i, x] := máx{V[i − 1, x], V[i − 1, x − w[i]] + v[i]}
        end for
    end for
    return V[n, W]
end procedure
```

Mochila Entera: Algoritmo de PD

- El coste de algoritmo es claramente $\mathcal{O}(nW)$
- Dicho coste es pseudopolinomial, ya que para expresar W solo se necesitan $\Theta(\log W)$ bits.

Mochila Entera: Algoritmo de PD

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

		w										
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	6	7	7	7	7	7	7	7	7	7
I	3	0	1	6	7	7	18	19	24	25	25	25	25
	4	0	1	6	7	7	18	22	23	28	29	29	40
	5	0	1	6	7	7	18	22	28	29	34	35	40

Ejemplo

$$v[4, 10] = \text{máx}\{v[3, 10], v[3, 10 - 6] + 22\} = \text{máx}\{25, 7 + 22\} = 29$$

$$v[5, 11] = \text{máx}\{v[4, 11], v[4, 11 - 7] + 28\} = \text{máx}\{40, 4 + 28\} = 40$$

Mochila Entera: Reconstrucción de la solución

Para encontrar el subconjunto $S \subseteq \{1, \ldots, n\}$ que alcanza el beneficio máximo, agregaremos una matriz K de tamaño $(n+1) \times (W+1)$ de tal modo que $K[i, x] =$ **true** si y solo si el objeto i está incluído en la solución con beneficio óptimo, es decir, si $v[i, x] = v[i-1, x-w[i]] + v[i]$.

procedure KNAPSACK(i, x)
    for $i := 0$ **to** n **do**
        $V[i, 0] := 0$; $K[i, 0] :=$ **false**;
    end for
    for $x := 1$ **to** W **do**
        $V[0, x] := 0$; $K[0, x] :=$ **false**;
    end for
    for $i := 1$ **to** n **do**
        for $x := 1$ **to** W **do**
            if
$V[i-1, x] \geq V[i-1, x-w[i]] + v[i]$ **then**
                $V[i, x] := V[i-1, x]$;
                $K[i, x] :=$ **false**;
            else

$V[i, x] := V[i-1, x-w[i]] + v[i]$;
                $K[i, x] :=$ **true**;
            end if
        end for
    end for
    return $\langle V, K \rangle$
end procedure

Mochila Entera: Reconstrucción de la solución

Ejemplo

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F
1	0 F	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T
2	0 0	1 0	6 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 T	23 T	28 T	29 T	29 T	40 T
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 T	29 T	34 T	35 T	40 0

Mochila Entera: Reconstrucción de la solución

Para calcular la solución óptima usamos la matriz K para recosntruirla, desde el final al principio. Si $K[n, W] =$ **true** entonces ponemos n en la solución óptima y saltamos a $K[n-1, W-w[n]]$ para ver si $n-1$ también está incuído o no, y así sucesivamente. Si $K[n, W] =$ **false** entonces n **no** está en la solución óptima y seguimos en $K[n-1, W]$.

```
x := W, S := ∅
for i := n downto 1 do
    if K[i, x] then
        S := S ∪ {i}
        x := x − w[i]
    end if
end for
return S
```

Coste: $\mathcal{O}(n + W)$

Mochila Entera: Reconstrucción de la solución

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F	0 F
1	0 F	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T	1 T
2	0 0	1 0	6 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T	7 T
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 T	23 T	28 T	29 T	29 T	40 T
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 T	29 T	34 T	35 T	40 0

$K[5, 11] \to K[4, 11] \to K[3, 5] \to K[2, 0]$. So $S = \{4, 3\}$

Parte IV

Programación Dinámica

- Introducción a la Programación Dinámica
- Mochila Entera
- Distancia de edición y otros problemas sobre strings
- Caminos Mínimos en Grafos

Distancia de edición

Dados dos strings $x = x_1 \cdots x_m$ e $y = y_1 \cdots y_n$, la distancia de edición entre ambos, $\text{dist}(x, y)$, es el mínimo número de operaciones de edición (inserción de un carácter, borrado de un carácter, sustitución de un carácter por otro) necesarias para convertir x en y.

Por ejemplo, $\text{dist}(\texttt{BARCO}, \texttt{ARCOS}) = 2$, pues hay que borrar la `B` inicial y agregar una `S` al final para pasar de un string al otro.

Otro ejemplo: $\text{dist}(\texttt{PALAS}, \texttt{ATRAS}) = 3$, ya que hay que borrar la `P` inicial, sustituir la `L` por `T` (o por `R`) e insertar una `R` (o una `T`).

1. Empezaremos estableciendo la recurrencia para la distancia de edición. Sea $\delta_{i,j}$ la distancia de edición entre el prefijo $x_1 \cdots x_i$ de longitud i de x y el prefijo $y_1 \cdots y_j$ de longitud j de y, $0 \leq i \leq m$, $0 \leq j \leq n$. Entonces la distancia buscada es $\text{dist}(x, y) = \delta_{m,n}$.
   La base de recursión es simple:

$$\begin{aligned} \delta_{0,j} &= j, \qquad 0 \leq j \leq n, \\ \delta_{i,0} &= i, \qquad 0 \leq i \leq m, \end{aligned}$$

   puesto que hay que insertar j caracteres para convertir la cadena vacía en $y_1 \cdots y_j$, y análogamente, hay que borrar i caracteres para convertir $x_1 \cdots x_i$ en la cadena vacía.

Distancia de edición

1. Para el caso general, la distancia de edición será uno de las tres siguientes posibilidades:
   - usar el mínimo de operaciones para convertir $x_1 \cdots x_{i-1}$ en $y_1 \cdots y_j$ y luego borrar x_i, ó
   - usar el mínimo de operaciones para convertir $x_1 \cdots x_i$ en $y_1 \cdots y_{j-1}$ y luego insertar y_j, ó
   - usar el mínimo de operaciones para convertir $x_1 \cdots x_{i-1}$ en $y_1 \cdots y_{j-1}$ y sustituir, si es necesario, x_i por y_j.

Distancia de edición

1. Poniendo todo junto, hay que tomar la opción que minimiza la distancia:

$$\delta_{i,j} = \text{mín}(\delta_{i-1,j} + 1, \delta_{i,j-1} + 1, \delta_{i-1,j-1} + s(x_i, y_j)),$$

donde $s(x_i, y_j) = 0$ si $x_i = y_j$ y $s(x_i, y_j) = 1$ si $x_i \neq y_j$.

Distancia de edición

2. Usaremos una matriz o tabla bidimensional D con $m+1$ filas por $n+1$ columnas de manera que $D[i,j] = \delta_{i,j}$.
La base de la recursión nos permite rellenar la fila 0 y la columna 0 de la tabla D. Por otro lado en la recurrencia obtenida en el paso anterior observamos que el valor $\delta_{i,j}$ depende del valor en la columna inmediatamente anterior $(i, j-1)$ y de dos de los valores de la fila previa $(i-1, j)$ e $(i-1, j-1)$. Esto significa que la matriz D debe rellenarse por filas de arriba $(i = 1)$ a abajo $(i = m)$ y, para cada fila, por columnas, de izquierda $(j = 1)$ a derecha $(j = n)$.

Distancia de edición

procedure EDITDIST(x, y)
▷ $m = |x|$, $n = |y|$, D: matriz $[0..m, 0..n]$ de enteros
  for $j := 0$ **to** n **do**
    $D[0, j] := j$
  end for
  for $i := 1$ **to** m **do**
    $D[i, 0] := i$
  end for
  for $i := 1$ **to** m **do**
    for $j := 1$ **to** n **do**
      ▷ $s(a, b)$ retorna 0 si $a = b$, y 1 si $a \neq b$

$$D[i, j] := \min\big(D[i-1, j] + 1, D[i, j-1] + 1,$$
$$D[i-1, j-1] + s(x[i], y[j])\big)$$

    end for
  end for
  return $D[m, n]$
end procedure

Distancia de edición

3. El coste del algoritmo viene naturalmente dominado por el coste del bucle principal, en el que se hacen $m \cdot n$ iteraciones, cada una de las cuales tiene coste $\Theta(1)$. El coste en espacio y en tiempo del algoritmo es $\Theta(m \cdot n)$.

   Para fijar ideas, si $m = c \cdot n$ entonces el tamaño de la entrada (= suma de las longitudes de los strings) es $m + n = \Theta(n)$, y el coste del algoritmo en tiempo y espacio es $\Theta(n^2)$, es decir, cuadrático respecto al tamaño de la entrada.

Distancia de edición

4. Resulta evidente que no necesitamos tener una matriz D completa: para hallar la entrada (i, j) nos basta tener las entradas previas de la fila i y la fila $i - 1$. Podemos entonces usar un par de vectores $D[0..n]$ y $Dprev[0..n]$ de manera que mantengamos el siguiente invariante: en la iteración que ha de calcular $\delta_{i,j}$ se cumple que $Dprev[k] = \delta_{i-1,k}$ para toda k y $D[k] = \delta_{i,k}$ para toda $k < j$.
   El coste del algoritmo en tiempo sigue siendo $\Theta(m \cdot n)$, pero el coste en espacio lo hemos reducido a $\Theta(n)$. Si $n > m$ convendrá intercambiar x e y al inicio: la distancia de edición es la misma.

Distancia de edición

procedure EDITDIST(x, y)
▷ $m = |x|$, $n = |y|$, $D, Dprev$: vectores $[0..n]$ de enteros
  for $j := 0$ **to** n **do**
    $Dprev[j] := j$
  end for
  for $i := 1$ **to** m **do**
    $D[0] := i$
    for $j := 1$ **to** n **do**

$$D[j] := \text{mín}\big(Dprev[j] + 1, D[j-1] + 1,$$
$$Dprev[j-1] + s(x[i], y[j])\big)$$

    end for
    ▷ $D \equiv \delta_{i,\cdot}$, $Dprev \equiv \delta_{i-1,\cdot}$
    $Dprev := D$ ▷ Copia D en $Dprev$
  end for
  return $D[n]$
end procedure

Subsecuencia Común Más Larga

En el problema de la subsecuencia común más larga (*longest common subsequence*, LCS) s nos dan dos secuencias X e Y y el objetivo es hallar un valor $k \geq 0$ máximo e índices $i_1 < i_2 < \cdots < i_k$ y $j_1 < \cdots < j_k$ tales que $x_{i_1} = y_{j_1}$, $x_{i_2} = y_{i_2}$, ...

Subsecuencia Común Más Larga

Sean $X = x_1 \cdots x_n$ e $Y = y_1 \cdots y_m$, y sea Z una subsecuencia común de X e Y de máxima longitud.

- Existen $i_1 < \cdots < i_k$ y $j_1 < \cdots < j_k$ tales que $Z = x_{i_1} \ldots x_{i_k} = y_{j_1} \ldots y_{j_k}$
- No existen i y j con $i > i_k$ y $j > j_k$ tales que $x_i = y_j$. Análogamente no existe i y j con $i < i_1$ y $j < j_1$ tales que $x_i = y_j$. En caso contrario Z no sería una LCS.
- $a = x_{i_k}$ puede aparecer tras i_k en X ($a = x_i$ para $i > i_k$), pero no tras la posición j_k en Y, y viceversa.
- Existe una LCS en la cual $a = x_{i_k} = y_{j_k}$ y no hay ninguna ocurrencia adicional de a ni en X ni en Y.

Subsecuencia Común Más Larga

1. Si $x[n] = Y[m]$ entonces podemos considerar que $i_k = n$ y $j_k = m$ (no sabemos todavía cuánto es k) pero $Z' = x_{i_1} \cdot x_{i_2} \cdots x_{i_{k-1}} = y_{j_1} \cdot y_{j_2} \cdots y_{j_{k-1}}$ es una LCS de $X[1..n-1]$ e $Y[1..m-1]$ y la longitud de una LCS de X e Y será 1 más la longitud de una LCS entre $X[1..n-1]$ e $Y[1..m-1]$.
2. Si $x[n] \neq Y[m]$ buscaremos LCS que involucren a $X[n]$ y no a $Y[m]$, a $Y[m]$ pero no a $X[n]$, y sin ninguno de los dos símbolos; y nos quedarenos con la más larga.

Subsecuencia Común Más Larga

1. Esto es, si $X[n] \neq Y[m]$ se calculará la longitud k_1 de una LCS correspondiente a $X[1..n]$ e $Y[1..m-1]$, la longitud k_2 de una LCS para $X[1..n-1]$ e $Y[1..m]$, y finalmente la longitud k_3 de la LCS para $X[1..n-1]$ e $Y[1..m-1]$. Entonces la longitud de la LCS para $X[1..n]$ e $Y[1..m]$ será $\text{máx}\{k_1, k_2, k_3\}$. Pero $k_3 \leq k_1$ o $k_3 \leq k_2$, por lo que solo hay que considerar dos de las tres posibilidades.
2. Sea $\lambda_{i,j}$ la longitud de una LCS entre $X[1..i]$ e $Y[1..j]$. Tenemos por lo tanto, si $i \neq 0$ y $j \neq 0$.

$$\lambda_{i,j} = \begin{cases} 1 + \lambda_{i-1,j-1} & \text{si } x_i = y_j, \\ \text{máx}\{\lambda_{i-1,j}, \lambda_{i,j-1}\} & \text{si } x_i \neq y_j. \end{cases}$$

Para los casos bases, $\lambda_{0,j} = \lambda_{i,0} = 0$ para cualesquiera i, j.

Subsecuencia Común Más Larga

- Construiremos una tabla L de tamaño $(n+1) \times (m+1)$ de manera que $L[i,j] = \lambda_{i,j}$. La longitud de una LCS entre X e Y vendrá dada en $L[n,m]$.
- La fila 0 y la columna 0 se inicializan con la base de la recurrencia: $L[i,0] := 0$ y $L[0,j] := 0$ para toda i y j.
- Para poder calcular $L[i,j]$ necesitamos tener $L[i-1,j]$, $L[i,j-1]$ y $L[i-1,j-1]$ previamente. Esto significa que debemos tener la fila $i-1$ completamente rellena y las primeras $j-1$ columnas de la fila i. En otras palabras L se rellena de izquierda a derecha ($j=1$ a $j=m$) y de arriba a abajo ($i=1$ a $i=n$)

Subsecuencia Común Más Larga

```
// retorna la longitud de una LCS de x e y
// asumimos que x[0] e y[0] no se usan
int LCS(const string& x, const string& y) {
  int n = x.length(); int m = y.length();
  // L = matrix (n+1) x (m+1)
  vector< vector<int> > L(n+1, vector<int>(m+1));
  for (int i = 0; i < n; ++i) L[i][0]= 0;
  for (int j = 0; j < m; ++j) L[0][j]= 0;
  for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
      if (x[i] == y[j])
         L[i][j] = L[i-1][j-1]+1;
      else
         L[i][j] = max(L[i-1][j], L[i][j-1]);
    }
  }
  return L[n][m];
}
```

Subsecuencia Común Más Larga

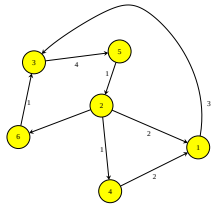
- El coste de esta solución en espacio y tiempo es claramente $\Theta(n \cdot m)$: hay $\Theta(n \cdot m)$ subproblemas a resolver y gracias a la recurrencia cada uno de ellos ser resuelve en tiempo $\Theta(1)$.
- Para obtener una LCS, podemos tener una estructura de datos auxiliar R de manera que $R[i][j]$ indica si $L[i][j]$ es $L[i-1][j]$, $L[i][j-1]$ o $L[i-1][j-1]+1$. En el último caso el último símbolo en la LCS de $X[1..i]$ e $Y[1..j]$ será $a = x[i][j]$. A continuación nos moveremos a la posición (i', j') que $R[i][j]$ nos indica.
- El coste en tiempo de la reconstrucción es $\mathcal{O}(n+m)$ pues en cada paso disminuye i o bien j o bien ambos en una unidad. El espacio auxiliar podemos ahorrarlo pues examinando $L[i-1][j]$, $L[i][j-1]$ y $L[i-1][j-1]$ es fácil determinar qué posición (i, j') ha determinado el valor $L[i][j]$.

Parte IV

Programación Dinámica

- Introducción a la Programación Dinámica
- Mochila Entera
- Distancia de edición y otros problemas sobre strings
- Caminos Mínimos en Grafos

Algoritmo de Floyd



Robert W. Floyd (1936–2001) Stephen Warshall (1935–2006)

El algoritmo de Floyd (1962) para los caminos mínimos entre todos los pares de vértices de un grafo dirigido G se basa en el esquema de PD. En realidad estamos resolviendo n^2 problemas de optimización, uno por cada par de vértices, pero están relacionados unos con otros y los resolveremos simultáneamente construyendo un espacio de estados adecuado. Concretamente consideramos el (sub)problema "camino mínimo de i a j que pase exclusivamente por los primeros k vértices de G (excluídos los extremos)". Sea $P_{i,j}^{(k)}$ la distancia de dicho camino.

Algoritmo de Floyd

Para solución $k = 0$ la solución es trivial: si $i = j$ entonces el coste del caamino mínimo es $P_{i,i}^{(0)} = 0$; si $i \neq j$ y $(i, j) \in E$ entonces el coste es el del arco que une i con j; si $i \neq j$ y $(i, j) \notin E$ entonces el coste es $+\infty$ ya que no existe un camino entre i y j usando cero vértices intermedios.

La respuesta buscada son los valores $P_{i,j}^{(n)}$, es decir, las distancia miínimas entre todos los pares de vértices (i, j), usando cualesquiera vértices.

Algoritmo de Floyd

La recurrencia para $P_{i,j}^{(k)}$ si se conoce $P_{u,v}^{(k-1)}$ para todo par de vértices (u, v). En efecto, el mejor camino de i a j será uno que no usa el vértice k $(= P_{i,j}^{(k-1)})$ o bien un camino que va de i a k y de k a j $(= P_{i,k}^{(k-1)} + P_{k,j}^{(k-1)})$. Fijoos en el uso del principio de optimalidad, de manera bastante intuitiva, en este problema: si π es un camino óptimo entre u y v que pasa por un vértice w entonces los subcaminos que van de u a w y de w a v son necesariamente mínimos. Si no lo fueran, entonces π no sería el mínimo!
Formalmente, si $k > 0$,

$$P_{i,j}^{(k)} = \text{mín}\{P_{i,j}^{(k-1)}, P_{i,k}^{(k-1)} + P_{k,j}^{(k1)}\}$$

Algoritmo de Floyd

A priori parece necesitarse una tabla de $\Theta(n^3)$ entradas para resolver el problema, pero puesto que los $P_{i,j}^{(k)}$'s dependen exclusivamente de los valores $P_{r,s}^{(k-1)}$ podríamos mantener exclusivamente dos tablas con n^2 entradas que mantengan los costes de las "fases" $k-1$ y k. El orden en que se resolviesen los n^2 subproblemas de la fase k sería irrelevante.

Algoritmo de Floyd

Pero podemos solucionar el problema con una sola tabla $n \times n$ ya que durante la fase k los únicos valores que necesitamos para evaluar el coste $P_{i,j}^{(k)}$ son el valor previo $P_{i,j}^{(k-1)}$ y los de la fila k ($P_{k,j}^{(k-1)}$) y la columna k ($P_{i,k}^{(k-1)}$).
Pero durante la fase k no puede cambiar ningún valor de la fila ni la columna k ya que $P_{k,k}^{(k-1)} = 0$.
Utilizaremos además otra tabla U de tamaño $n \times n$ para registrar cuáles son los caminos mínimos. La entrada $U[i,j] = 0$ si el camino mínimo entre i y j no existe o consiste en un único arco que une i y j. Por otra parte, si $U[i,j] = k > 0$, $(i \neq k, j \neq k)$ entonces el camino se compone de dos caminos óptimos: el que va de i a k y el que va de k a j.
El procedimiento que permite recuperar el camino mínimo entre dos vértices a partir de la información en U es sencillo.

Algoritmo de Floyd

```
typedef vector < vector<double> > AdjMatrix;
typedef AdjMatrix MinCostMatrix;
typedef vector< vector<int> > MinPathMatrix;
// Pre:
// G[i][j] == peso del arco (i,j) si existe,
//         == +infinity si el arco (i,j) si no existe
// Post: ver la descripcion del texto
void floyd(const AdjMatrix& G, MinCostMatrix& P,
           MinPathMatrix& U) {
  int n = G.size();
  P = G;
  for (int i = 0; i < n; ++i) P[i][i] = 0;
  U = MinPathMatrix(n, vector<int>(n, -1));
  for (int k = 0; k < n; ++k)
     for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
           if (P[i][k]+P[k][j] < P[i][j]) {
             P[i][j] = P[i][k]+P[k][j];
             U[i][j] = k;
           }
}
```

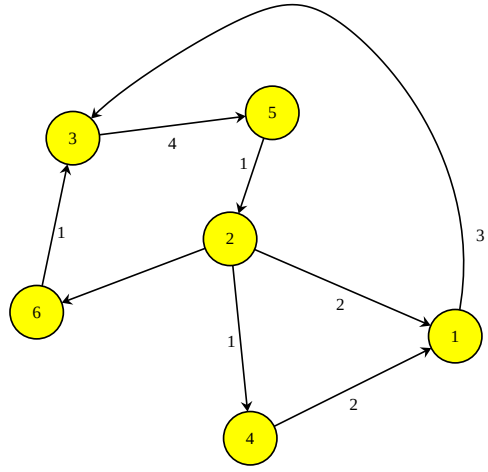
Algoritmo de Floyd

El coste en espacio del algoritmo de Floyd es $\Theta(n^2)$. Por otra parte, es obvio que el coste en tiempo es $\Theta(n^3)$. Una alternativa al uso de este algoritmo consiste en emplear el algoritmo de Dijkstra, con un vértice distinto como fuente en cada ocasión. El coste de esta alternativa, usando listas de adyancencias y un *heap* para seleccionar el candidato de cada fase es, en caso peor, $\Theta(mn \log n)$, donde m es el número de arcos.

Si el grafo es denso ($m = \Omega(n^2)$) entonces el algoritmo de Floyd será preferible por su menor coste asintótico y su gran sencillez. Si el grafo es disperso ($m \ll n^2$) será más conveniente emplear n veces el algoritmo de Dijkstra.

Si el algoritmo de Dijkstra no utilizase el *heap* y el grafo viniese representado mediante una matriz de adyacencia entonces el coste de n-Dijkstra sería $\Theta(n^3)$ y el algoritmo de Floyd resultaría más atractivo por su manifiesta simplicidad.

Algoritmo de Floyd



$$P^{(0)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & \infty & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

Algoritmo de Floyd

$$P^{(1)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & 5 & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & \infty & \infty \\ \infty & 1 & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

$$P^{(2)} = \begin{pmatrix} 0 & \infty & 3 & \infty & \infty & \infty \\ 2 & 0 & 5 & 1 & \infty & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & \infty & \infty \\ 3 & 1 & 6 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

$$P^{(4)} = P^{(3)} = \begin{pmatrix} 0 & \infty & 3 & \infty & 7 & \infty \\ 2 & 0 & 5 & 1 & 9 & 2 \\ \infty & \infty & 0 & \infty & 4 & \infty \\ 2 & \infty & 5 & 0 & 9 & \infty \\ 3 & 1 & 6 & 2 & 0 & 3 \end{pmatrix}$$

Algoritmo de Floyd

$$P^{(5)} = \begin{pmatrix} 0 & 8 & 3 & 9 & 7 & 10 \\ 2 & 0 & 5 & 1 & 9 & 2 \\ 7 & 5 & 0 & 6 & 4 & 7 \\ 2 & 10 & 5 & 0 & 9 & 12 \\ 3 & 1 & 6 & 2 & 0 & 3 \\ 8 & 6 & 1 & 7 & 5 & 0 \end{pmatrix}$$

$$P^{(6)} = \begin{pmatrix} 0 & 8 & 3 & 9 & 7 & 10 \\ 2 & 0 & 3 & 1 & 7 & 2 \\ 7 & 5 & 0 & 6 & 4 & 7 \\ 2 & 10 & 5 & 0 & 9 & 12 \\ 3 & 1 & 4 & 2 & 0 & 3 \\ 8 & 6 & 1 & 7 & 5 & 0 \end{pmatrix}$$

Algoritmo de Floyd

Si en vez de inicializar la matriz P con los costes trabajamos con la matriz de adyacencia de G y cambiamos el cálculo del $\min$ por $\vee$ y $+$ por $\wedge$, el algoritmo resultante es el denominado algoritmo de Warshall para el cálculo de la clausura transitiva de G. Es decir, al finalizar $P[i,j] = |true|$ si y sólo si existe un camino entre i y j en G. No es un algoritmo de PD propiamente dicho ya que no estamos optimizando ningún coste.

```
typedef vector< vector<bool> > AdjMatrix;
void Warshall(AdjMatrix& G) {
  int n = G.size();
  for (int k = 0; k < n; ++k)
     for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
           G[i][j] = G[i][j] or (G[i][k] and G[k][j]);
}
```