Algorítmica
Programación Dinámica

Conrado Martínez
U. Politècnica Catalunya

ALG Q2-2024–2025

Temario

Parte I: Repaso de Conceptos Algorítmicos

Parte II: Búsqueda y Ordenación Digital

Parte III: Algoritmos Voraces

Parte IV: Programación Dinámica

Parte V: Flujos sobre Redes y Programación Lineal

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
3. Caminos Mínimos en Grafos

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

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:

- El problema de la mochila entera.
- Cálculo de la distancia de edición entre cadenas.
- Cálculo de la cadena de multiplicación de matrices óptima.
- 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 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).

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

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

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

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

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

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

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

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 ≠ b
            D[i, j] := mín(D[i − 1, j] + 1, D[i, j − 1] + 1,
                           D[i − 1, j − 1] + s(x[i], y[j]))
        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] := mín(Dprev[j] + 1, D[j − 1] + 1,
                        Dprev[j − 1] + s(x[i], y[j]))
        end for
        ▷ D ≡ δ_{i,·}, Dprev ≡ δ_{i−1,·}
        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

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Weighted Activity Selection

Tenemos n actividades que no pueden ser ejecutadas simultáneamente; cada actividad tienen un tiempo de inicio s_i, un tiempo de finalización f_i y un peso w_i. El objetivo es seleccionar un subconjunto de actividades compatibles cuyo peso combinado es máximo.
En el tema de Algoritmos Voraces vimos que no había una forma de resolver este problema con una estrategia voraz.

Weighted Activity Selection

Supongamos que

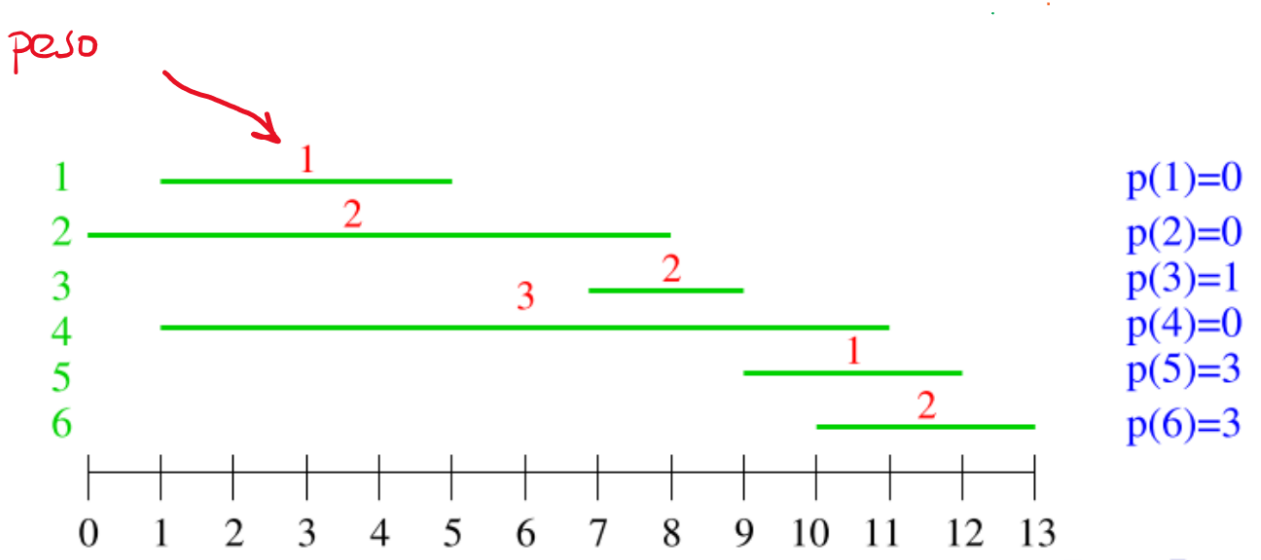
$$f_1 \leq f_2 \leq \cdots f_n$$

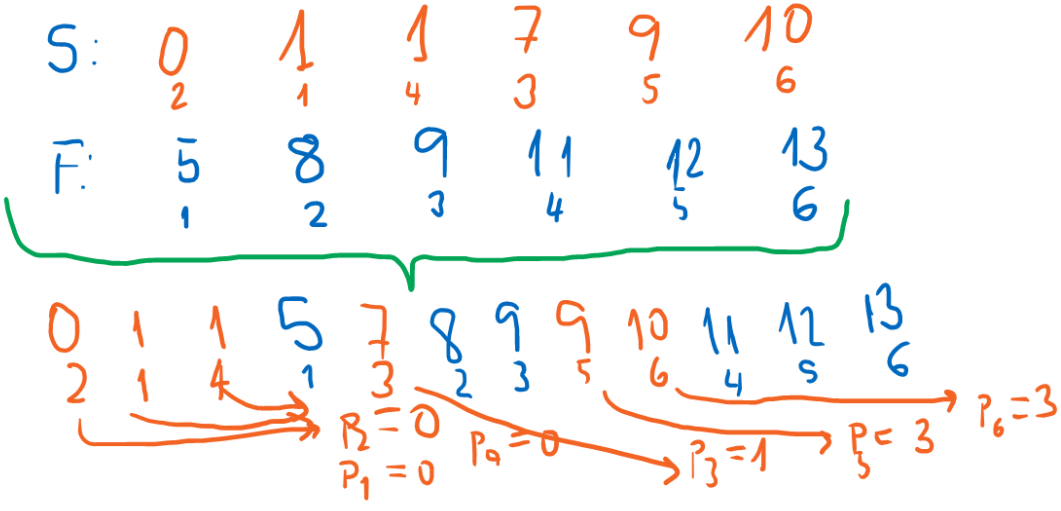
Podemos ordenar el conjunto de actividades en tempo $\mathcal{O}(n \log n)$ de modo que esto se cumpla.

Dado un valor i definimos p_i el mayor índice $j < i$ de una actividad compatible con la actividad i, esto es, el mayor j tal que $f_j \leq s_i$; si no existe, $p_i = 0$.

Si ordenamos los tiempo de inicio en orden creciente (coste: $\mathcal{O}(n \log n)$) podemos usar una variación de la búsqueda binaria para calcular p_i con coste $\mathcal{O}(\log n)$ para toda i; también podemos calcular todos los p_i's con coste $\mathcal{O}(n)$ "fusionando" la lista ordenada de tiempos de finalización con la lista ordenada de tiempos de inicio.

Weighted Activity Selection





Weighted Activity Selection

Sea W_i el peso máximo alcanzable con las actividades $\{1, \ldots, i\}$. Claramente $W_1 = w_1$, ya que no hay otras actividades. Si $i \leq 0$, no hay actividades que puedan ser seleccionadas y por convenio tomaremos $W_i = 0$ si $i \leq 0$. En general, si $i > 1$ podemos considerar dos opciones:

1. La actividad i forma parte de la solución óptima; entonces las actividades $p_i + 1, \ldots, i - 1$ **no** pueden formar parte de la solución porque son incompatibles con la actividad i, y el resto de la solución tiene que ser óptima para las actividades $\{1, \ldots, p_i\}$, esto es, $W_i = w_i + W_{p_i}$
2. La actividad i no forma parte de la solución óptima, luego la solución óptima lo es para las actividades $\{1, \ldots, i - 1\}$ y $W_i = W_{i-1}$

La recurrencia de PD es por lo tanto:

$$W_i = \text{máx}\{W_{i-1}, w_i + W_{p_i}\}, \quad i \geq 1$$
$$W_i = 0, \quad i < 1$$

Weighted Activity Selection

Para resolver el problema solo necesitamos memorizar los valores de W_i en una tabla, $1 \leq i \leq n$ y rellenarla de menor a mayor índice, con coste lineal: para cada i, solo necesitamos coste $\mathcal{O}(1)$ para rellenar $W[i] \equiv W_i$.

```
// weight[1..n] = pesos de las actividades 1 a n
// s[1..n] = tiempos de inicio
// f[1..n] = tiempos de finalizacion; f[1] <= f[2] <= ... <= f[n]
// calcular  p[1..n]
// p[i] = mayor indice j < i de una actividad compatible con la i
//        (f[j] <= s[i])
vector<double> W(n+1,0);
vector<bool> sel(n+1, false);
for (int i = 1; i <= n; ++i) {
  W[i] = W[i-1];
  if (weight[i] + W[p[i]] > W[i]) {
    W[i] = weight[i] + W[p[i]];
    sel[i] = true;
  }
}
// sel[i] = true <=> la actividad i está incluída en
//                   la solución óptima
// W[n] = peso total de la solución óptima
```

Weighted Activity Selection

El coste total del algoritmo es $\mathcal{O}(n \log n)$.

1. Coste $\mathcal{O}(n \log n)$ para ordenar las actividades por f_i creciente, y por otro lado, por s_i creciente
2. Coste $\mathcal{O}(n)$ para calcular el vector p fusionando las dos secuencias ordenadas del paso previo
3. Coste $\mathcal{O}(n)$ de la PD propiamente dicha, incluyendo la construcción de la solución

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Multiplicación de matrices

Ejemplo

Supongamos que tenemos 3 matrices $A_{m\times n}$, $B_{n\times p}$ y $C_{p\times r}$. El producto $D = A \cdot B \cdot C$ es una matriz $m \times r$ y por la propiedad asociativa del producto de matrices podemos calcularlo como $D = (A \cdot B) \cdot C$ o como $D = A \cdot (B \cdot C)$.
Usando el algoritmo simple para multiplicar las matrices, en el primer caso haremos $mnp + mpr$ multiplicaciones; en el segundo haremos $npr + mnr$. Supongamos que $m = 20$, $n = 10$, $p = 15$ y $r = 4$; la primera alternativa necesita 4200 multiplicaciones, la segunda solo 1400.

Multiplicación de matrices

- Nuestro objetivo será determinar la forma óptima de parentizar las multiplicaciones. En general tendremos n matrices $A_1, A_2, \ldots, A_n$ siendo $f_i \times c_i$ las dimensiones de A_i; necesariamente $f_i = c_{i-1}$ para toda i, $1 < i \leq n$.
- La lista de valores $[c_0 = f_1, c_1, c_2, \ldots, c_n]$ especifica completamente el problema.

Multiplicación de matrices

Supongamos que queremos calcular el producto de $A_i, \ldots, A_j$ de manera óptima.

- Si $i = j$, solo tenemos una matriz y no hay nada que hacer, se necesitan 0 operaciones
- Si $i < j$ entonces el producto se puede dividir

$$(A_i \times \cdots \times A_k) \cdot (A_{k+1} \times \cdots \times A_j)$$

y el número de operaciones es: 1) el número de operaciones para calcular $A_i \times \cdots \times A_k$; + 2) el número de operaciones para calcular $A_{k+1} \times \cdots \times A_j$; + 3) $f_i c_k c_j = c_{i-1} c_k c_j$ para multiplicar la primera parentización por la segunda

Multiplicación de matrices

Si definimos $C_{i,j}$ como el número mínimo de operaciones para realizar el producto $A_i \times \cdots \times A_j$ obtenemos

$$C_{i,j} = \begin{cases} 0 & \text{if } i \geq j, \\ \min_{i \leq k < j} \{C_{i,k} + C_{k+1,j} + c_{i-1} c_k c_j\} & \text{if } i < j. \end{cases}$$

Multiplicación de matrices

Para la memoización y la versión iterativa del algoritmo de PD nos fijamos en:

1. solo hay que rellenar la matriz triangular superior
2. para calcular la componente $C[i, j]$ que ha de contener el valor $C_{i,j}$ necesitamos tener calculados los valores de la fila i previos a la columna j (esto es, $C_{i,i}$, $C_{i,i+1}$, ..., $C_{i,j-1}$) y también los valores de la columna j por debajo de la fila i (es decir $C_{i+1,j}$, $C_{i+2,j}$, ..., $C_{j,j}$)

Conviene por lo tanto rellenar la diagonal principal (es un caso base) y después rellenar de abajo hacia arriba y de izquierda a derecha (alternativamente por diagonales de SE a NW)

Multiplicación de matrices

```
// col[i] = columnas de la matriz i-ésima, 1 <= i <= n
// col[0] = filas de la primera matriz
vector<vector<int>> C(n+1, vector<int>(n+1));
for (int i = 1; i <= n; ++i) C[i][i] = 0;
for (int i = n; i >= 1; --i) {
  for (int j = i+1; j <= n; ++j) {
    C[i][j] = C[i+1][j] + col[i-1]*col[i]*col[j];
    for (int k = i+1; k < j; ++k) {
       int opk = C[i][k]+C[k+1][j]+col[i-1]*col[k]*col[j];
       if (opk < C[i][j]) C[i][j] = opk;
    }
  }
}
```

Multiplicación de matrices

La solución obtenida tiene coste $\Theta(n^3)$ en tiempo y $\Theta(n^2)$ en espacio.
Para hallar la parentización guardaremos en $P_{i,j}$ el valor k con el que se alcanza el valor óptimo $C_{i,j}$ ($P_{i,i} = i$ por convenio).

```
      C[i][i] = 0; ¿{\color{red}\texttt{P[i][i] = i;}}¿
...
C[i][j] = C[i+1][j] + col[i-1]*col[i]*col[j];
¿{\color{red}\texttt{P[i][j] = i;}}¿
for (int k = i+1; k < j; ++k) {
   int opk = C[i][k]+C[k+1][j]+col[i-1]*col[k]*col[j];
   if (opk < C[i][j]) {
     C[i][j] = opk; ¿{\color{red}\texttt{P[i][j] = k;}}¿
   }
}
...
```

Multiplicación de matrices

La matriz P “codifica” la estructura arborescente de la parentización óptima: si $P[1,n] = k$ el producto $A_1 \times \cdots \times A_n$ debe hacerse como $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$. A su vez el producto $(A_1 \times \cdots \times A_k)$ se hará óptimamente como $(A_1 \times \cdots \times A_{P_{1,k}}) \times (A_{P_{1,k}+1} \times \cdots \times A_k)$ y así sucesivamente.

Multiplicación de matrices

```
Matriz hacer_productos(const vector<Matriz>& A,
                       const vector<vector<int>>& P, int i, int j) {
    if (i == j) return A[i];
    else {
      Matriz X = hacer_productos(A, P, i, P[i][j]);
      Matriz Y = hacer_productos(A, P, P[i][j]+1, j);
      return X * Y; // producto matricial
    }
}
```

El árbol de llamadas recursivas nos da la parentización óptima y se hacen un total de $C_{1,n}$ productos; el algoritmo recursivo hace un total de n llamadas recursivas.

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Árboles binarios de búsqueda óptimos

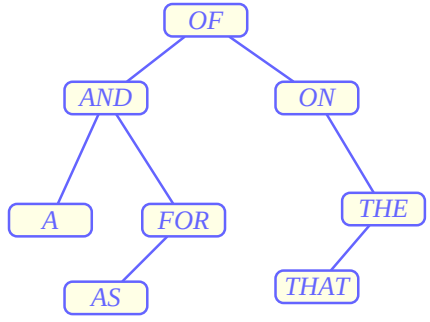
Supongamos que tenemos n elementos $x_1, \ldots, x_n$ con $x_1 < x_2 < \cdots < x_n$ y supongamos que conocemos las respectivas probabilidades de acceso o consulta $p_1, p_2, \ldots, p_n$. Obviamente, $\sum_{1 \leq i \leq n} p_i = 1$. Parece natural organizar los elementos de tal modo que sea eficiente el acceso a aquéllos con mayor probabilidad de ser consultados.
Sea T un árbol binario de búsqueda con los n elementos. El *coste medio de acceso* de T es:

$$C_{\mathbf{p}}(T) = 1 + \sum_{1 \leq i \leq n} p_i \cdot \text{PROF}(T, x_i),$$

donde $\text{PROF}(T, x_i)$ denota la profundidad de x_i en T.
El problema que nos planteamos es: dado un conjunto $X = \{x_1 < \cdots < x_n\}$ y $\mathbf{p} = (p_1, \ldots, p_n)$ construir el árbol binario T^* para X tal que $C_{\mathbf{p}}(T^*)$ es mínimo.

Árboles binarios de búsqueda óptimos

x_i	f_i
A	142
AND	58
FOR	129
OF	266
ON	93
THE	180
TO	32
	900



$$C(T_{eq}) = 1930/900 \approx 2{,}14$$

$$C(T^*) = 1904/900 \approx 2{,}11$$

Árboles binarios de búsqueda óptimos

Supongamos que x_k ocupa la raíz de T^*. Entonces resulta evidente que su subárbol izquierdo es necesariamente óptimo para el conjunto $\{x_1 < \cdots < x_{k-1}\}$ con probabilidades $\mathbf{p}_{1,k-1} = (p_1/w_{1,k-1}, \ldots, p_{k-1}/w_{1,k-1})$ donde $w_{1,k-1} = p_1 + \cdots + p_{k-1}$. Análogamente, el subárbol derecho de T^* es óptimo para el subconjunto $\{x_{k+1}, \ldots, x_n\}$ con probabilidades $\mathbf{p}_{k+1,n} = (p_{k+1}/w_{k+1,n}, \ldots, p_n/w_{k+1,n})$, donde $w_{k+1,n} = p_{k+1} + \cdots + p_n$. Es decir, se cumple el principio de optimalidad.

Sea $w_{i,j} = p_i + \cdots + p_j$. Consideraremos como peso de un árbol el producto de su coste por la suma de los pesos de los elementos que contiene.

Árboles binarios de búsqueda óptimos

Sea $T_{i,j}$ el árbol de peso óptimo y $c_{i,j} = C(T_{i,j})$. Supongamos que x_k, $i \leq k \leq j$, es la raíz de $T_{i,j}$. Entonces

$$
\begin{aligned}
w_{i,j}c_{i,j} &= w_{i,j} + \sum_{i \leq r \leq j} p_r \cdot \mathrm{PROF}(T_{i,j}, x_r) \\
&= w_{i,j} + \sum_{i \leq r < k} p_r \cdot (1 + \mathrm{PROF}(T_{i,k-1}, x_r)) \\
&\quad + \sum_{k < r \leq j} p_r \cdot (1 + \mathrm{PROF}(T_{k+1,j}, x_r)) \\
&= w_{i,j} + w_{i,k-1} + \sum_{i \leq r < k} p_r \mathrm{PROF}(T_{i,k-1}, x_r) \\
&\quad + w_{k+1,j} + \sum_{k < r \leq j} p_r \mathrm{PROF}(T_{k+1,j}, x_r) \\
&= w_{i,j} + w_{i,k-1}c_{i,k-1} + w_{k+1,j}c_{k+1,j}.
\end{aligned}
$$

Árboles binarios de búsqueda óptimos

Obviamente, la ecuación recursiva es

$$w_{i,j}c_{i,j} = w_{i,j} + \min_{i \leq k \leq j} \left(w_{i,k-1}c_{i,k-1} + w_{k+1,j}c_{k+1,j} \right),$$

con el convenio de que $w_{i,j} = 0$ si $i > j$. Los subproblemas, en este caso, son de la forma (i, j) con $1 \leq i \leq j \leq n$. Los casos base son de la forma (i, i) con $1 \leq i \leq n$.

Árboles binarios de búsqueda óptimos

Sólo nos queda por hallar una manera adecuada de resolver los subproblemas. Recorriendo la "matriz" triangular superior de izquierda a derecha y de abajo hacia arriba garantizamos que cuando intentemos resolver el problema (i, j) ya tendremos resueltos todos aquéllos de los que depende. Otra forma de hacerlo es por diagonales en el sentido arriba-izquierda a abajo-derecha. La primera diagonal, formada por los estados (i, i) tiene solución trivial. Después vienen las diagonales de los estados $(i, i + 1)$, $(i, i + 2)$, etc.

Árboles binarios de búsqueda óptimos

El algoritmo comienza rellenando una matriz auxiliar $W[1..n, 1..n]$ con los pesos $w_{i,j}$, y a continuación los "pesos" $d[i,j] = w_{i,j}c_{i,j}$ de acuerdo con la estrategia comentada. El coste espacial es $\Theta(n^2)$ mientras que el coste temporal es claramente $\Theta(n^3)$. Además de calcular la matriz de $d_{i,j}$'s el procedimiento determina las raíces en otra matriz adicional $root$ a partir de la cual se puede reconstruir el árbol óptimo en tiempo $\Theta(n)$. Obsérvese que el coste buscado $c_{1,n} = d[1,n]$ ya que $w_{1,n} = 1$.

Árboles binarios de búsqueda óptimos

```
procedure BST_OPTIMO(p, d, root)
    Matrix<double> W(0..n+1, 1..n);
    for i := 1 to n+1 do
        W[i, i-1] := 0; d[i, i-1] := 0
        for j := i to n do
            W[i, j] := W[i, j-1] + p[j]
        end for
    end for
    ...
end procedure
```

Árboles binarios de búsqueda óptimos

```
procedure BST_OPTIMO(p, d, root)
    ...
    for i := n downto 1 do
        j := i;
        while j ≤ n do
            min := +∞;
            for k := i to j do
                cc := d[i, k − 1] + d[k + 1, j]
                if cc < min then
                    min := cc; root[i, j] := k
                end if
            end for
            d[i, j] := W[i, j] + min
            j := j + 1
        end while
    end for
end procedure
```

Árboles binarios de búsqueda óptimos

En 1971, Knuth observó que
$root[i, j-1] \leq root[i, j] \leq root[i+1, j]$ para todo $1 \leq i < j \leq n$.
Ello permite restringir el rango en la búsqueda del valor de k que minimiza $d_{i,k-1} + d_{k+1,j}$.

```
for k := root[i, j-1] to root[i+1, j] do
    cc := d[i, k-1] + d[k+1, j]
    if cc < min then
        min := cc; root[i, j] := k
    end if
end for
```

A fin de tratar adecuadamente el caso de base $i = j$ conviene inicializar $root[i, i-1] = i$ en el bucle de inicialización.
Con este sencillo cambio el coste del algoritmo baja de $\Theta(n^3)$ a $\Theta(n^2)$. Este tipo de mejora se puede aplicar en bastantes situaciones y con ello se consigue rebajar en un factor n el coste temporal de algoritmos de PD.

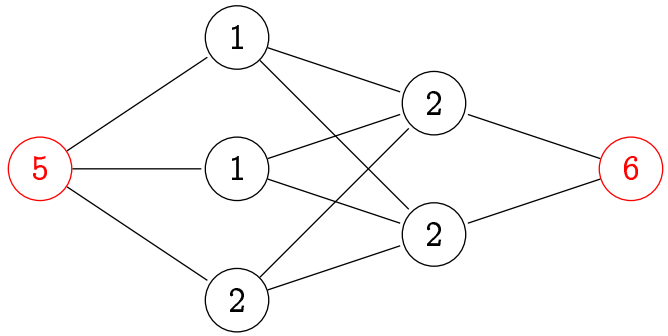
Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Maximum Weight Independent Set (MWIS)

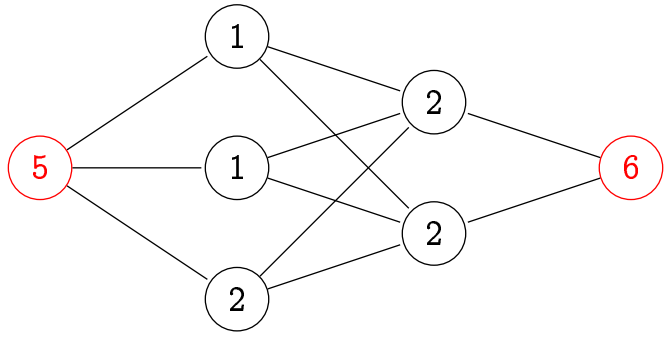
Dado un grafo $G = \langle V, E \rangle$ con pesos en los vértices $w : V \to \mathbb{R}$ el objetivo es hallar un subonjunto $S \subseteq V$ tal que no existe ninguna arista $(u, v) \in E$ entre vértices $u, v \in S$ y el peso total de S es máximo.



Para grafos generales el problema es NP-duro, inclusive en el caso en que todos los pesos valen 1 (y por tanto se trata de maximizar la cardinalidad de S—problema del *Maximum Independent Set* (MIS).

Maximum Weight Independent Set (MWIS)

Dado un grafo $G = \langle V, E \rangle$ con pesos en los vértices $w : V \to \mathbb{R}$ el objetivo es hallar un subonjunto $S \subseteq V$ tal que no existe ninguna arista $(u, v) \in E$ entre vértices $u, v \in S$ y el peso total de S es máximo.

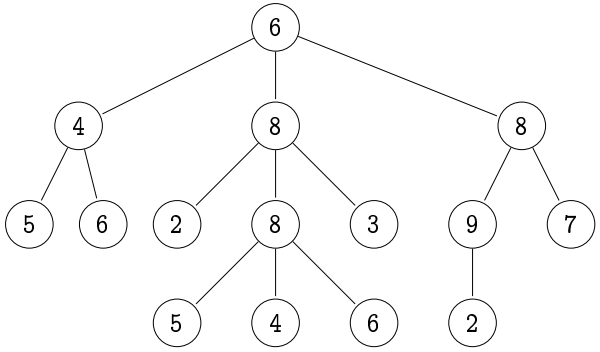


Para grafos generales el problema es NP-duro, inclusive en el caso en que todos los pesos valen 1 (y por tanto se trata de maximizar la cardinalidad de S—problema del *Maximum Independent Set* (MIS).

Maximum Weight Independent Set en Árboles

Dado un árbol T y un vértice $r \in V$ denotamos T_r al árbol enraizado en r.

Dado un árbol enraizado $T_r = \langle V, E, r \rangle$ con pesos $w : V \to \mathbb{R}$ hallar el MWIS de T_r



Notación:

- Dado un árbol T_r y un vértice $v \in V$ denotamos $C(v)$ el conjunto de hijos de v, y $G(v)$ el conjunto de nietos de v.
- Denotamos por S_v^* al MWIS de T_v y $M(v) = \sum_{x \in S_v} w(x)$

Maximum Weight Independent Set en Árboles

Observación: sea S_v un *independent set* de T_r y x un elemento de S_v. Entonces ni el padre de x en T_r ni ninguno de los hijos de x puede pertenecer a S_v.

Consideremos S^*, una solución óptima del MWIS en $T \equiv T_r$

1. $r \in S^*$: entonces $C(r) \not\subseteq S^*$. Por lo tanto $S^* - \{r\}$ contiene soluciones óptimas de cada uno de los T_v donde $v \in G(r)$, i.e., $S^* = \{r\} \cup \bigcup_{v \in G(r)} S_v^*$
2. $r \notin S^*$: entonces S^* contiene soluciones óptimas de cada uno de los T_v donde $v \in C(r)$, i.e., $S^* = \bigcup_{v \in C(r)} S_v^*$

Maximum Weight Independent Set en Árboles

Observación: sea S_v un *independent set* de T_r y x un elemento de S_v. Entonces ni el padre de x en T_r ni ninguno de los hijos de x puede pertenecer a S_v.
Consideremos S^*, una solución óptima del MWIS en $T \equiv T_r$

1. $r \in S^*$: entonces $C(r) \not\subseteq S^*$. Por lo tanto $S^* - \{r\}$ contiene soluciones óptimas de cada uno de los T_v donde $v \in G(r)$, i.e., $S^* = \{r\} \cup \bigcup_{v \in G(r)} S_v^*$
2. $r \notin S^*$: entonces S^* contiene soluciones óptimas de cada uno de los T_v donde $v \in C(r)$, i.e., $S^* = \bigcup_{v \in C(r)} S_v^*$

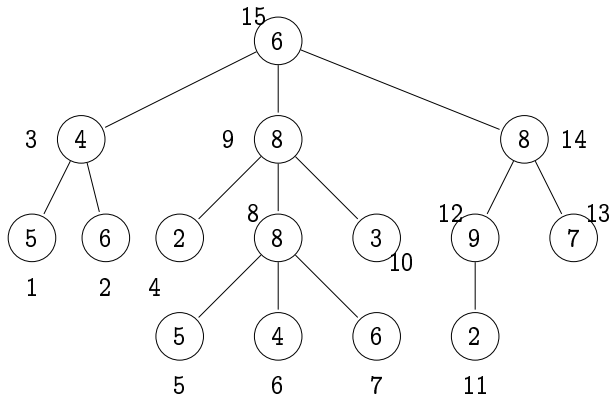
Maximum Weight Independent Set en Árboles

Para implementar la PD, para cada nodo v vamos guardando $\langle v, M(v) \rangle$ en una tabla asociativa (un *map*). Recordar que $M(v)$ es el peso de la solución óptima enraizada en T_v.

También resultará conveniente tener otra tabla asociativa que guarde $M'[v] = \sum_{x \in C(v)} M(x)$

Maximum Weight Independent Set en Árboles

Para obtener los valores $M(v)$ y $M'(v)$ se hace un recorrido postorden del árbol.



Maximum Weight Independent Set en Árboles

```
procedure MWIS(T)
    v := raiz de T
    if v es una hoja then
        ▷ C(v) = ∅
        M[v] := WEIGHT(v); M'[v] := 0
        return M[v]
    else
        M'[v] := 0; μ := 0
        for u ∈ C(v) do
            MWIS(T_u)
            M'[v] := M'[v] + M[u]; μ := μ + M'[u]
        end for
        M[v] := máx{μ + WEIGHT(v), M'[v]}
        return M[v]
    end if
end procedure
```

Coste: espacio $\mathcal{O}(n)$; tiempo $\mathcal{O}(n)$

Maximum Weight Independent Set en Árboles

Con un segundo recorrido del árbol podemos marcar los vértices que forman parte de la solucíon óptima, empezando desde la raíz; en cada llamada recursiva se pasa un booleano que indica si el padre forma parte o no del MWIS.
Si $M[r] = M'[r]$ entonces r no forma parte del MWIS, ponemos $inMWIS[r] :=$ **false** y se hace llamada recursiva en cada subárbol con el booleano **false**. Si $M[r] > M'[r]$ hacemos $inMWIS[r] :=$ **true** y llamada recursiva en cada subárbol con el booleano **true**.

En general en un vértice v si la llamada es con el booleano **true**, hacemos $inMWIS[v] :=$ **false** y llamada recursiva en cada subárbol con el booleano **false**. En caso contrario se compara $M[v]$ con $M'[v]$ y se da valor a $inMWIS$ y se hacen las llamadas recursivas según que $M[v] = M'[v]$ o $M[v] > M'[v]$.

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Caminos Mínimos en Grafos

En este tema estudiaremos algoritmo para hallar caminos mínimos en grafos dirigidos ponderados $G = \langle V, E \rangle$ donde tenemos un peso $w : E \to \mathbb{R}$ en cada arco.

- El peso $w(\pi)$ de un camino $\pi = u = v_0, v_1, \ldots, v_\ell = v$ de longitud $\ell \geq 0$ es la suma de los pesos de los arcos (v_i, v_{i+1}) que forman el camino.
- Si el grafo G tiene arcos de peso negativo entonces puede contener ciclos de peso negativo y en consecuencia habrá pares de vértices (u, v) para los cuales la noción de camino de peso mínimo no está bien definida
- Si existe un camino de u a v en G ($u \rightsquigarrow_G v$) definimos

$$\Delta(u, v) = \text{mín}\{w(\pi) \mid \pi \text{ es un camino de } u \text{ a } v\}$$

y $\Delta(u, v) = +\infty$ si $u \not\rightsquigarrow_G v$. Denotamos $\pi^*(u, v)$ a un camino de peso mínimo $\Delta(u, v)$ entre u y v.

Caminos Mínimos en Grafos

En este tema estudiaremos algoritmo para hallar caminos mínimos en grafos dirigidos ponderados $G = \langle V, E \rangle$ donde tenemos un peso $w : E \to \mathbb{R}$ en cada arco.

- El peso $w(\pi)$ de un camino $\pi = u = v_0, v_1, \ldots, v_\ell = v$ de longitud $\ell \geq 0$ es la suma de los pesos de los arcos (v_i, v_{i+1}) que forman el camino.
- Si el grafo G tiene arcos de peso negativo entonces puede contener ciclos de peso negativo y en consecuencia habrá pares de vértices (u, v) para los cuales la noción de camino de peso mínimo no está bien definida
- Si existe un camino de u a v en G ($u \rightsquigarrow_G v$) definimos

$$\Delta(u, v) = \text{mín}\{w(\pi) \mid \pi \text{ es un camino de } u \text{ a } v\}$$

y $\Delta(u, v) = +\infty$ si $u \not\rightsquigarrow_G v$. Denotamos $\pi^*(u, v)$ a un camino de peso mínimo $\Delta(u, v)$ entre u y v.

Caminos Mínimos en Grafos

En este tema estudiaremos algoritmo para hallar caminos mínimos en grafos dirigidos ponderados $G = \langle V, E \rangle$ donde tenemos un peso $w : E \to \mathbb{R}$ en cada arco.

- El peso $w(\pi)$ de un camino $\pi = u = v_0, v_1, \ldots, v_\ell = v$ de longitud $\ell \geq 0$ es la suma de los pesos de los arcos (v_i, v_{i+1}) que forman el camino.
- Si el grafo G tiene arcos de peso negativo entonces puede contener ciclos de peso negativo y en consecuencia habrá pares de vértices (u, v) para los cuales la noción de camino de peso mínimo no está bien definida
- Si existe un camino de u a v en G ($u \rightsquigarrow_G v$) definimos

$$\Delta(u, v) = \text{mín}\{w(\pi) \mid \pi \text{ es un camino de } u \text{ a } v\}$$

y $\Delta(u, v) = +\infty$ si $u \not\rightsquigarrow_G v$. Denotamos $\pi^*(u, v)$ a un camino de peso mínimo $\Delta(u, v)$ entre u y v.

Caminos Mínimos en Grafos

Consideraremos dos tipos de problemas:

1. Single Source Shortest Paths (SSSP): hallar todos los caminos de peso mínimo en G desde un vértice s dado (la fuente) a los restantes; se calcula $\Delta(s, u)$ para toda $u \in V$
2. All Pairs Shortest Paths (APSP): hallar los caminos de peso mínimo en G entre todo par de vértices; se calcula $\Delta(u, v)$ para todo par $(u, v) \in V \times V$

El problema APSP puede resolverse invocando $n = |V|$ veces un algoritmo que resuelve SSSP, usando como fuente cada uno de los n vértices de G en cada llamada.

Caminos Mínimos en Grafos

	SSSP		APSP		
	Dijkstra	BF	FW	n-Dijkstra	Johnson
$w \geq 0$	$\mathcal{O}(m + n \log n)^{(1)}$	$\mathcal{O}(nm)$	$\Theta(n^3)$	$\mathcal{O}(nm + n^2 \log n)$	$\mathcal{O}(nm + n^2 \log n)^{(1)}$
$w \in \mathbb{R}$	no	$\mathcal{O}(nm)$	$\Theta(n^3)$	no	$\mathcal{O}(mn + n^2 \log n)^{(1)}$

(1) = Usando un Fibonacci heap
BF = Bellman-Ford
FW = Floyd-Warshall

Los algoritmos de Bellman-Ford y de Johnson además detectan la existencia de ciclos de peso negativo

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Algoritmo de Bellman-Ford

Richard E. Bellman (1920–1984)

Lester R. Ford Jr (1927–2017)

Es similar al algoritmo de Dijkstra, se mantiene como invariante que para todo vértice $D[v]$ es una cota superior a la distancia mínima desde s a v:

$$D[v] \geq \Delta(s, v)$$

Además, si $D[v] \neq +\infty$ entonces $path[v]$ es el predecesor de v en un camino de peso $D[v]$ entre s y v.

Algoritmo de Bellman-Ford

El algoritmo de Dijkstra mantiene ese mismo invariante y adicionalmente que $D[v] = \Delta(s, v)$ si v es un vértice `visitado`; en cada iteración se visita un vértice, por lo cual después de n iteraciones del bucle principal se han determinado caminos mínimos entre s y los restantes vértices del grafo.
El ingrediente fundamental en el algoritmo de Dijkstra y también el de Bellman-Ford es la regla de relajación (*relax rule*):

if $D[v] > D[u] + w(u, v)$ **then**
    $D[v] := D[u] + w(u, v)$
    $path[v] := u$
end if

Algoritmo de Bellman-Ford

Si se aplica la regla de relajación el invariante se mantiene:

$$\Delta(s,v) \leq \Delta(s,u) + w(u,v) \leq D[u] + w(u,v)$$

$\uparrow$ por la desigualdad triangular + h.i.

$$\Delta(s,v) \leq D[v] \quad \text{h.i.}$$

$$\Longrightarrow \Delta(s,v) \leq \min(D[v], D[u] + w(u,v))$$

Algoritmo de Bellman-Ford

procedure BELLMAN-FORD(G, w, s)
  for $v \in V(G)$ **do**
    $D[v] := +infty$; $path[v] := \perp$
  end for
  $D[s] := 0$; $path[s] := s$;
  for $i := 1$ **to** $n - 1$ **do**
    for $(u, v) \in E(G)$ **do**
      RELAX(u, v)
    end for
  end for
  for $(u, v) \in E(G)$ **do**
    if $D[v] > D[u] + w(u, v)$ **then** ▷ Negative cycle!!
    end if
  end for
  return $\langle D, path \rangle$
end procedure
▷ Coste: $\Theta(|V| \cdot |E|)$

Algoritmo de Bellman-Ford

Lema

Para todo vértice v, tras la i-ésima iteración de BF

$D[v] \leq \min(w(P) \mid P$ *es un camino de* s *a* v *de longitud* $\leq i)$

Demostración

Por inducción sobre i. La base de la inducción es correcta antes de entrar en el bucle para $i = 0$. Si es cierto para $i-1$ iteraciones entonces al aplicar la relajación sobre todos los arcos (u, v) del digrafo el invariante será cierto ya que para todo vértice el bucle interno acaba considerando todos los caminos de longitud $i-1$ hasta v y los de longitud $i-1$ para todo predecesor de v. Luego $D[v]$ será a lo sumo la distancia mínima de s a v mediante un camino de longitud $\leq i$. □

Algoritmo de Bellman-Ford

Teorema

1. *Si G no contiene ciclos de peso negativo, al terminar BF se cumple $D[v] = \Delta(s, v)$ para todo v*
2. *Si G contiene ciclos de peso negativo, BF lo reportará.*

Algoritmo de Bellman-Ford

Demostración

1. Si no hay ciclos negativos, $\Delta(s, v)$ está bien definido para todo v y viene dado por un camino simple de longitud $\leq n - 1$: no puede usar más arcos que vértices sin repetir y no hay ninguna ventaja en que contenga un ciclo (porque no hay ciclos negativos); $D[v] \geq \Delta(s, v)$ por el invariante del algoritmo y $D[v] \leq \Delta(s, v)$ por el lema anterior, luego $D[v] = \Delta(s, v)$.
2. Si hay ciclos negativos, al menos uno de los arcos del ciclo puede ser relajado siempre, y el último bucle de BF justamente lo detecta

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Caminos mínimos en DAGs

Los caminos mínimos en un DAG desde s a los demás vértices tienen una estructura simple y están siempre bien definidos (ya que por definición no hay ciclos!):

1. Si v no es accesible desde s, $\Delta(s, v) = +\infty$
2. Si $v = s$ entonces $\Delta(s, s) = 0$ y el camino mínimo es el camino vacío
3. Si $s \rightsquigarrow v$ y $v \neq s$ entonces un camino mínimo de s a v consiste en un camino mínimo de s a un predecesor u de v, seguido del arco (u, v):

$$\Delta(s, v) = \min\{\Delta(s, u) + w(u, v) \mid u \text{ es un predecesor de } v\}$$

Caminos mínimos en DAGs

Para calcular los caminos mínimos basta recorrer los arcos siguiendo el orden topológico inverso del subgrafo G' de vértices accesibles desde s.

```
procedure MinDist(G, s)
    for v ∈ V do D[v] := +∞; path[v] := ⊥
    end for
    D[s] := 0; path[s] := s;
    for v ∈ Successors(G, s) do
        MinDist-Rec(G, s, v)
    end for
end procedure
```

Caminos mínimos en DAGs

```
procedure MinDist-Rec(G, u, v)
    ▷ Actualizar D[v] teniendo en cuenta
    ▷ el camino mínimo de s a u
    Relax(u, v)
    for w ∈ Successors(G, v) do
        MinDist-Rec(G, v, w)
    end for
end procedure
```

Coste: $\Theta(|V| + |E|)$ (es un DFS!)

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

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 considerando un conjunto subproblemas adecuado.

Algoritmo de Floyd

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/peso de dicho camino.

Para $k = 0$ la solución también es trivial: si $i = j$ entonces el coste del camino mínimo es $P_{i,i}^{(0)} = 0$; si $i \neq j$ y $(i, j) \in E$ entonces el peso es el del arco que une i con j; si $i \neq j$ y $(i, j) \notin E$ entonces el peso 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 $P_{i,j}^{(k)}$ es simple si se conoce $P_{u,v}^{(k-1)}$ para todo par de vértices (u, v). En efecto, el mejor camino de i a j usando alguno(s) de los vértices $\{1, \ldots, k\}$ como vértices intermedios 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 $(= P_{i,k}^{(k-1)} + P_{k,j}^{(k-1)})$, usando alguno(s) de los primeos $k - 1$ vértices.

Fijaos 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!

Algoritmo de Floyd

Formalmente,

$$P_{i,j}^{(k)} = \begin{cases} 0 & \text{si } k = 0 \text{ y } i = j \\ +\infty & \text{si } k = 0,\ i \neq j \text{ y } (i,j) \notin E \\ w(i,j) & \text{si } k = 0,\ i \neq j \text{ y } (i,j) \in E \\ \text{mín}\{P_{i,j}^{(k-1)}, P_{i,k}^{(k-1)} + P_{k,j}^{(k-1)}\} & \text{si } k > 0 \end{cases}$$

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;
           }
}
```

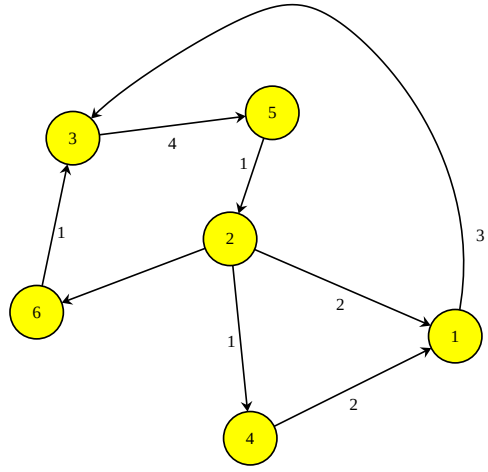
Algoritmo de Floyd

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 podría ser usar el algoritmo de Dijkstra, con un vértice distinto como fuente en cada ocasión. El coste de esta alternativa, usando listas de adyancencias y un *Fibonacci heap* para seleccionar el candidato de cada fase es, en caso peor, $\Theta(mn + n^2 \log n)$, donde m es el número de arcos.

Algoritmo de Floyd

Si el grafo es denso ($m = \Omega(n^2)$) entonces el algoritmo de Floyd será preferible por 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 *Fibonacci 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 seguría siendo más atractivo por su 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 Warshall

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;
// Pre: G[i][j] == true iff (i,j) es un arco en G
// Post: G[i][j] == true si hay camino de i a j
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]);
}
```

Parte IV

Programación Dinámica

1. Introducción a la Programación Dinámica
2. Ejemplos de Programación Dinámica
   - Mochila Entera
   - Distancia de edición y otros problemas sobre strings
   - Weighted Activity Selection
   - Multiplicación de Matrices
   - Construcción de BSTs Óptimos (**)
   - Maximum Weight Independent Set en Árboles
3. Caminos Mínimos en Grafos
   - Algoritmo de Bellman-Ford
   - Caminos mínimos en DAGs
   - Algoritmo de Floyd-Warshall
   - Algoritmo de Johnson

Algoritmo de Johnson

La idea del algoritmo de Johnson es sencilla:

1. Ejecutar el algoritmo de Bellman-Ford una vez para determinar si G tiene (o no) ciclos negativos; si los hay, reportarlo y acabar
2. Reescalar los pesos para que todos sean positivos, pero de manera que se preserven en G' los caminos mínimos del grafo original
3. Aplicar n veces el algoritmo de Dijkstra sobre G' y hacer la transformación inversa para obtener las distancias mínimas en G

El coste es:

$$\begin{aligned}\Theta(\text{BF} + n + m + n \times \text{DIJKSTRA}) &= \Theta(n \cdot m + n(m + n \log n)) \\ &= \Theta(nm + n^2 \log n)\end{aligned}$$

Algoritmo de Johnson

Lema

Sea G un digrafo ponderado con pesos $w : E \to \mathbb{R}$. Sea $f : V \to \mathbb{R}$ una función cualquiera y definimos $w_f(u,v) = w(u,v) + f(u) - f(v)$ para todo arco $(u,v) \in E$. Si π es un camino $u \rightsquigarrow v$ en G con peso $w(\pi)$ entonces $w_f(\pi) = w(\pi) + f(u) - f(v)$.

Algoritmo de Johnson

1. Dado el grafo original G añadimos un vértice ficticio s y arcos (s, v) con peso 0 para todo vértice v del grafo original, obteniendo un nuevo grafo $G' = \langle V', E', w' \rangle$
2. Ejecutamos BF con entrada (G', s). G' contiene ciclos negativos ssi G contiene ciclos negativos; si los detectamos, se termina el algoritmo.
3. Si no hay ciclos negativos, $D[v] = \Delta_{G'}(s, v)$.

Algoritmo de Johnson

4. Definimos $G_D = \langle V, E, w_D \rangle$ donde $w_D(u, v) = w(u, v) + D[u] - D[v]$
5. Se aplica Dijkstra n veces sobre G_D

Algoritmo de Johnson

Lema

Si $\pi^*(u, v)$ es un camino mínimo entre u y v en G, entonces también los es en G_D. Además

$$\Delta_{G_D}(u, v) = \Delta_G(u, v) + D[u] - D[v].$$

Demostración

Para todo camino π en G se cumple $w_D(\pi) = w(\pi) + D[u] - D[v]$. Solo el primer término en la expresión depende del camino π, de manera que si $w(\pi)$ es mínimo entonces $w_D(\pi)$ también, y viceversa. □

Algoritmo de Johnson

Lema

Todos los pesos en G_D son positivos.

Demostración

- Como G' no contiene ciclos negativos, se cumple la desigualdad triangular: para todo camino π de u a v

$$\Delta_{G'}(s,v) \leq \Delta_{G'}(s,u) + w(\pi)$$

- Luego

$$\Delta_{G'}(s,u) - \Delta_{G'}(s,v) + w(\pi) \geq 0$$
$$D[u] - D[v] + w(\pi) = w_D(\pi) \geq 0$$

□

Algoritmo de Johnson

procedure JOHNSON(G)
  Calcular G' ▷ coste: $\Theta(n)$
  $D :=$ BELLMAN-FORD(G', s) ▷ coste: $\Theta(n \cdot (m+n))$
  Calcular G_D ▷ coste: $\Theta(m)$
  for $u \in V$ **do** $d[u] :=$ DIJKSTRA(G_D, u)
  end for ▷ coste: $\Theta(n \cdot \text{Dijsktra}(n, m))$
  for $u \in V$ **do**
    ▷ Invertir transformación: $w_D(\pi) = w(\pi) + D[u] - D[v]$
    $d[u][v] := d[u][v] + D[v] - D[u]$
  end for ▷ coste: $\Theta(n)$
end procedure
▷ Coste total: $\Theta(mn + n(m + n\log n) = \Theta(nm + n^2 \log n)$