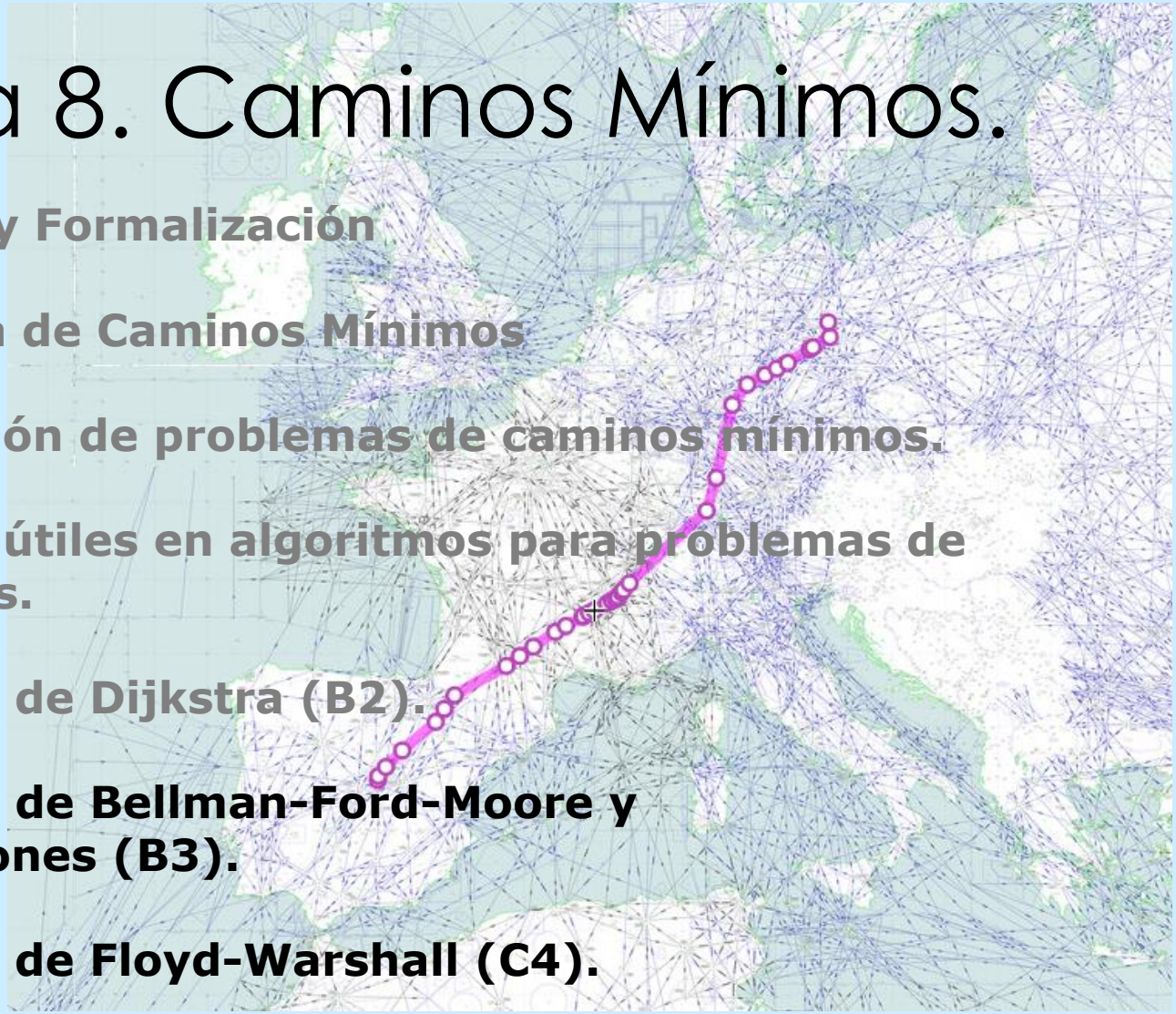
Tema 8. Caminos Mínimos.



8.1 Notación y Formalización

8.2 Existencia de Caminos Mínimos

8.3 Clasificación de problemas de caminos mínimos.

8.4 Etiquetas útiles en algoritmos para problemas de uno a todos.

8.5 Algoritmo de Dijkstra (B2).

8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3).

8.7 Algoritmo de Floyd-Warshall (C4).

Dpto. de Matemáticas, Estadística e Investigación Operativa. Universidad de La Laguna.

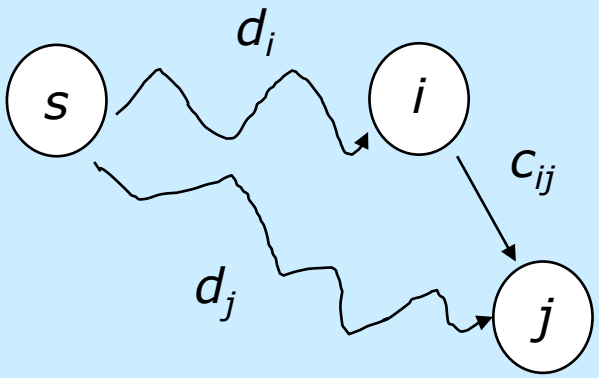
8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3

Problema B3. Determinar los caminos mínimos de uno-a-todos con costes arbitrarios, pero sumiendo que *G* no contiene circuitos de coste negativo. El algoritmo del estado-del-arte es el **algoritmo de Bellman-Ford-Moore**.

A diferencia con el problema B2, no podemos saber que etiquetas distancias son óptimas o no en cualquier momento. Por eso debemos pensar que condiciones deben cumplir cuando finalice el algoritmo, es decir, para que sean óptimas.

Condiciones de Optimalidad.

Al final del algoritmo, para todo arco (i , j) de A se debe cumplir que $d_j \leq d_i + c_{ij}$.



Si las etiquetas distancias son óptimas, no puede ser que el camino de s a j usando el arco (i, j) tenga menor coste, por tanto:

$$d_j \leq d_i + c_{ij} \quad \forall (i,j) \in A$$

8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3

La manera más eficiente de examinar si los arcos satisfacen las condiciones de optimalidad es la siguiente:

1) Se inicializan las etiquetas distancias a 0 para el nodo *s* y $+\infty$ la de los demás.
2) Sólo se examinan los nodos que recientemente mejoraron su etiqueta distancia (la primera vez es el nodo *s*). Aquí examinar significada que si la etiqueta de un nodo mejoró, examinamos sus sucesores para ver si podemos mejorar sus distancias (¿es $d_j > d_i + c_{ij}$?)

¿en qué orden se examinan los nodos? **En orden FIFO, es decir, primero en modificar su distancia, primero en examinar a sus sucesores. El orden FIFO se implementa de manera fácil mediante una cola.**

Al igual que en el recorrido en Amplitud, se asegura que se alcanza a cada nodo usando el camino de menor coste con el menor número de arcos.

Este es el Algoritmo de Bellman-Ford-Moore y corre en O(*nm*).

8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3

Note que un nodo puede cambiar su etiqueta distancia más de una vez (a lo sumo n-1 si no hay circuitos de longitud negativa). Por tanto, cada vez que su etiqueta distancia mejora, se añade a la cola si no está en ella.

La modificación siguiente asegura un algoritmo más rápido en la práctica:

Un nodo que ha estado en la cola, la siguiente vez que entre en la cola debe insertarse al principio ("se le permite colarse").

Está modificación es debida a Pape y D'sopo. Llamaremos a este algoritmo como PDM (Pape y D'sopo Modification).

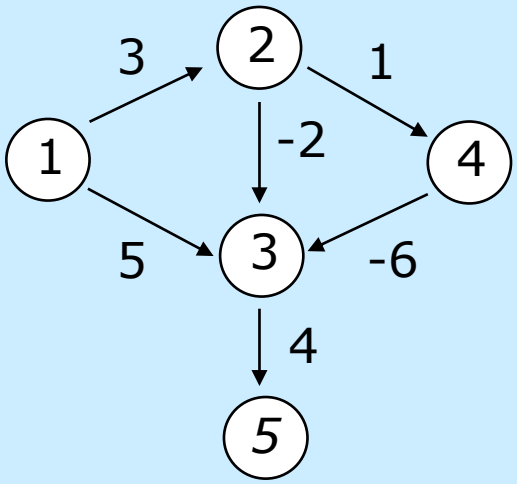
Para la implementación del algoritmo usaremos una ***cola Q*** con las siguientes funciones o métodos que requieren tiempo constante (O(1)):

1) *InicializarCola*(*Q*) que crea una cola vacía.
2) *ColaNoVacía*(*Q*) devuelve True si la cola no está vacía, en otro caso False.
3) *InsertarEnCola*(*i*, *Q*) añade al final de la cola el nodo *i*.
4) *IsertarPrincipioCola*(*i*, *Q*) añade al principio de la cola el nodo *i*.
5) *ExtraerDeCola*(*i, Q*) extrae el primer elemento de la cola y devuelve en *i* el correspondiente nodo.
6) *NoEstaEnCola*(i, Q) devuelve True si el nodo *i no* está en la cola, en otro caso False.

8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3). Pseudocó

```
Algoritmo PDM(G, s, var d, Var pred){
    //inicialización
    Para todo i = 1 hasta n {d_i = +∞; pred_i = 0;}
    InicializarCola(Q);
    d_s = 0; pred_s = s; InsertarEnCola(s,Q)
    //bucle
    Mientras (ColaNoVacía(Q)) Hacer {
        ExtraerDeCola(i,Q);
        //A continuación propagamos etiqueta de i
        Para todo nodo j sucesor del nodo i Hacer
            Si (d_j > d_i + c_ij) entonces{
                // si el nodo j nunca ha estado en la cola
                Si (pred_j == 0) entonces
                    InsertarEnCola(j,Q); // final de cola
                en otro caso // ya ha estado en la cola
                    si NoEstaEnCola(j,Q)entonces
                        InsertarPrincipioCola(j, Q);
                d_j = d_i + c_ij; pred_j = i;
            }
    }
}
```

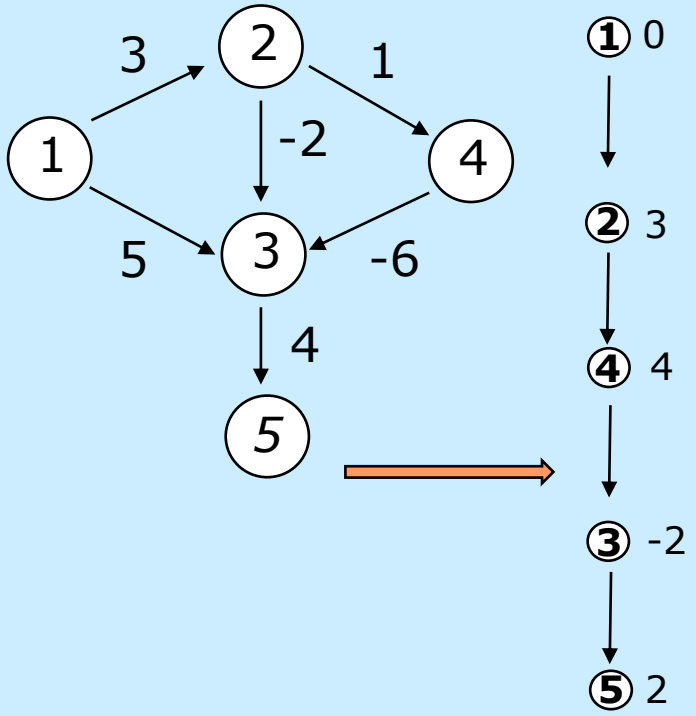
8.6 Algoritmo de Bellman-Ford-Moore y modificaciones (B3). Tra



La traza del algoritmo PDM es una tabla donde la primera columna (iter), refleja la iteración; la segunda columna es el estado del cola Q; la tercera columna el nodo *i* extraído de la cola y luego hay tantas columnas como nodos que reflejan el par (*d*, *pred*). Cada fila refleja la ejecución entera de una iteración. El origen es *s*=1. Uno nodo aparece en rojo en la cola cada vez que se cuela. En negrita la modificación de etiquetas.

Iter	***Cola Q***	***i***	***d, pred***				
			1	2	3	4	5
Ini	1		0, 1	+∞, 0	+∞, 0	+∞, 0	+∞, 0
1	2, 3	1	0, 1	**3**, 1	**5**, 1	+∞, 0	+∞, 0
2	3, 4	2	0, 1	3, 1	**1**, 2	**4**, 2	+∞, 0
3	4, 5	3	0, 1	3, 1	1, 2	4, 2	**5**, 3
4	3, 5	4	0, 1	3, 1	**-2**, 4	4, 2	5, 3
5	5	3	0, 1	3, 1	-2, 4	4, 2	**2**, 3
6		5	0, 1	3, 1	-2, 4	4, 2	2, 3

el árbol de caminos mínimos resultante**.**



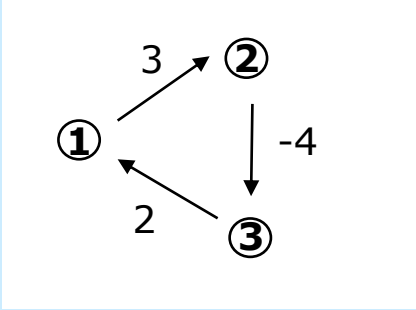
La complejidad del algoritmo es O(min{nC,2^n}), donde C es el máximo de los costes en valor absoluto. Es decir, podría requerir un número exponencial de examinaciones de los arcos. Esto es debido al eventual comportamiento como LIFO en vez de FIFO debido a la modificación. Se acepta el algoritmo por su buen comportamiento medio (en la práctica).

Ojo!!! Notar que el algoritmo no termina si existe un circuito de coste negativo. Habría que incorporar un mecanismo de detección. Existen varios en la Literatura.

8.7 Algoritmo de Floyd-Warshall (C4).

Problema C4. Determinar los caminos mínimos de todos-a-todos con G y costes arbitrarios. El algoritmo del estado-del-arte es el **algoritmo de Floyd y Warshall**.

Dado el siguiente grafo queremos calcular los caminos mínimos desde entre todos los pares de nodos.



Una solución consistiría en aplicar tres veces el PDM o Bellman-Ford-Moore, con *s* variando de 1 a 3, el resultado sería:

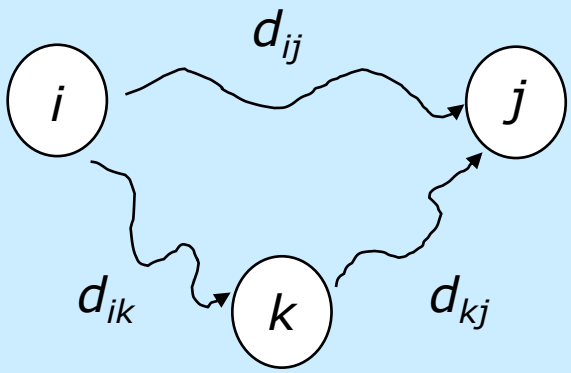
	d			pred		
s	1	2	3	1	2	3
1	0	3	-1	1	1	2
2	2	0	-4	3	2	2
3	2	5	0	3	1	3

Pero es más eficiente si usamos Floyd-Warshall, necesitamos un tiempo $O(n^3)$. El objetivo aquí es calcular la matriz de distancias mínimas d_{ij}.

8.7 Algoritmo de Floyd-Warshall (C4).

Condiciones de Optimalidad.

Desde que hay costes negativos, debemos pensar que deben satisfacer las distancias $\boldsymbol{d_{ij}}$ para que sean óptimas.



El camino de para ir de i a j debe ser menor o igual que la suma de los caminos mínimos para ir de *i* a *j* pasando por cualquier nodo intermedio *k* que no está en el primer camino:

$$d_{ij} \leq d_{ik} + d_{kj} \quad \forall i, j, k \in V \text{ con } i \neq k \text{ y } j \neq k$$

Una implementación directa de estas condiciones de optimalidad implicarían un algoritmo corriendo en $O(n^4)$.
Pero existe una forma más eficiente de hacerlo. Buscando el orden adecuado ⟶

8.7 Algoritmo de Floyd-Warshall (C4).

Ecuaciones de recurrencia.

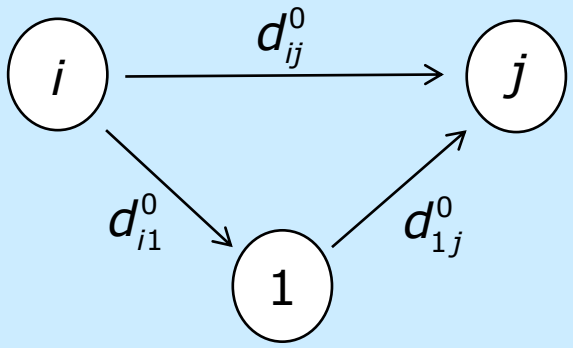
Define:

d_{ij}^0 = el coste mínimo del camino de i a j que no pasa por nodos intermedios, es decir,

$$d_{ij}^0 = \begin{cases} 0 & \text{si } i = j \\ c_{ij} & \text{si } i \neq j \text{ y } (i,j) \in A \\ +\infty & \text{si } i \neq j \text{ y } (i,j) \notin A \end{cases}$$

Ahora podemos definir:

d_{ij}^1 = el coste mínimo del camino de i a j que a lo sumo puede tener como nodos intermedios a nodos en el conjunto {1}, es decir,



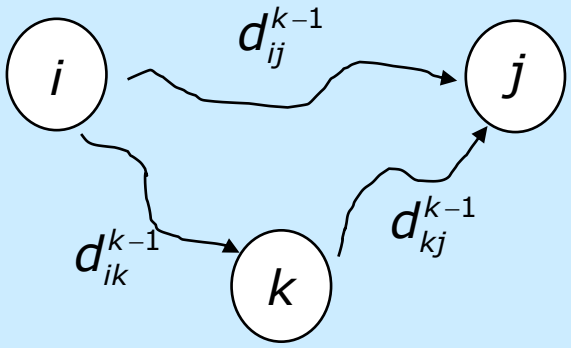
$$d_{ij}^1 = \min\left\{d_{ij}^0, d_{i1}^0 + d_{1j}^0\right\} \text{ con } i \neq 1 \text{ y } j \neq 1$$

Lo mejor entre ir directamente mediante el arco si existe o pasar exclusivamente por el nodo 1.

8.7 Algoritmo de Floyd-Warshall (C4).

Ecuaciones de recurrencia.

Ahora en la etapa k, definimos:
d_{ij}^k = el coste mínimo del camino de i a j que a lo sumo puede tener como nodos intermedios a nodos en el conjunto $\{1,2,\ldots,k\}$, es decir,



$$d_{ij}^k = \min\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\} \text{ con } i \neq k \text{ y } j \neq k$$

Lo mejor entre ir pudiendo pasar por los nodos en $\{1,\ldots,k\text{-}1\}$ o pudiendo también pasar adicionalmente por el nodo k.

Finalmente, en la etapa n, definimos:
d_{ij}^n = el coste mínimo del camino de i a j que a lo sumo puede tener como nodos intermedios a nodos en el conjunto $\{1,2,\ldots,n\}$, es decir, tendremos **las distancias óptimas**, pues habremos considerado la posibilidad de pasar por todos los nodos del grafo.

8.7 Algoritmo de Floyd-Warshall (C4).

Implementación de las Ecuaciones de recurrencia.

En una etapa genérica *k*, tenemos:

$$d_{ij}^{k} = \min\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\} \text{ con } i \neq k \text{ y } j \neq k$$

Lo que implica sólo usar dos matrices, la de la etapa *k* y la de la etapa anterior. Pero si observamos bien la ecuación de recurrencia, los únicos elementos (celdas) que no pueden cambiar son aquellos con *i=k y j=k* que precisamente son los argumentos que se usan para calcular la celdas en la etapa *k.* Por tanto, *una única matriz distancias es necesaria que se construye desde k=0 (base de la recurrencia) hasta n donde el superíndice no aparece, es implícito:*

Para todo $k = 1$ hasta n

$$d_{ij} = \min\left\{d_{ij}, d_{ik} + d_{kj}\right\} \ \forall i, j \in V \text{ con } i \neq k \text{ y } j \neq k$$

8.7 Algoritmo de Floyd-Warshall (C4).

Matriz de predecesores.

Al igual que en los problemas de tipo B, necesitamos una estructura para identificar los nodos y arcos en el camino mínimo.
En este caso hablamos de la **matriz de predecesores P_{ij}** que almacena el nodo anterior al nodo j en el camino (mínimo) de *i* a *j*. Se inicializa y actualiza de la manera siguiente:

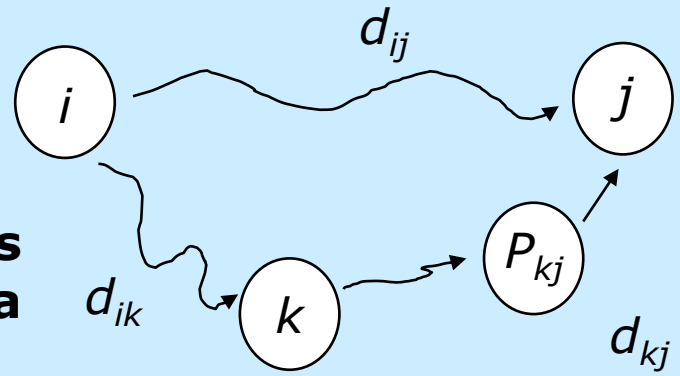
Para $k = 0$ (inicialización)

$$P_{ij} = \begin{cases} i & \text{si } i = j \\ i & \text{si } i \neq j \text{ y } (i,j) \in A \\ 0 & \text{si } i \neq j \text{ y } (i,j) \notin A \end{cases}$$

Para todo k> 0, se actualiza de la manera siguiente:

si $d_{ij} > d_{ik} + d_{kj}$ entonces $P_{ij} = P_{kj}$;
en otro caso no cambia

Así, fácilmente podemos usar los procedimientos *MostrarCaminos* ya vistos.

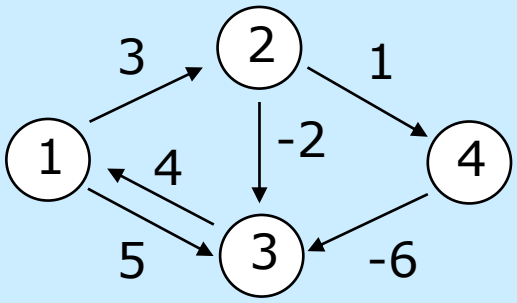


8.7 Algoritmo de Floyd-Warshall (C4). Pseudocódigo.

```
Algoritmo Floyd-Warshall(G, var d, var P){
    //inicialización
    Para todo i = 1 hasta n
        Para todo j = 1 hasta n
            Si (i == j) entonces {d_ij = 0; P_ij = i;}
            en otro caso
                Si (i,j) ∈ A entonces {d_ij = c_ij; P_ij = i;}
                en otro caso {d_ij = +∞; P_ij = 0;}
    //bucle
    Para todo k = 1 hasta n
        Para todo i = 1 hasta n
            Si (i ≠ k) entonces
                Para todo j = 1 hasta n
                    Si (j ≠ k) entonces
                        Si (d_ij > d_ik + d_kj) entonces
                            {d_ij = d_ik + d_kj; P_ij = P_kj;}
}
```

Es trivial observar que la complejidad es $O(n^3)$.

8.7 Algoritmo de Floyd-Warshall (C4). Traza.



La traza del algoritmo Floyd y Warshall es una secuencia de las matrices d y P para k=0 hasta n (aquí 4). Luego añadimos una columna para indicar la etapa *k*. Para todo $k>0$, la fila y la columna k no cambian. Estás se copiaran de las matriz anterior y son sombreadas en rojo. Las celdas que cambian, su valor aparece en negrita.

		d				***P***			
		1	2	3	4	1	2	3	4
$k=0$	1	0	3	5	∞	1	1	1	0
	2	∞	0	-2	1	0	2	2	2
	3	4	∞	0	∞	3	0	3	0
	4	∞	∞	-6	0	0	0	4	4

		d				***P***			
		1	2	3	4	1	2	3	4
$k=1$	1	0	3	5	∞	1	1	1	0
	2	∞	0	-2	1	0	2	2	2
	3	4	**7**	0	∞	3	**1**	3	0
	4	∞	∞	-6	0	0	0	4	4

Primero copiamos fila y columna 1 (k=1) en ambas matrices. Para cada celda restante comprobamos si $d_{ij} > d_{i1}+d_{1j}$ si es así hacemos $d_{ij} = d_{i1}+d_{1j}$ y $P_{ij} = P_{1j}$. En otro caso, copiamos lo que había en la celda de la matriz anterior. Pero, esto se automatiza como indican las flechas para no enredarnos con los sub-índices.

8.7 Algoritmo de Floyd-Warshall (C4). Traza.

		d				**P**			
		1	2	3	4	1	2	3	4
$k=1$	1	0	3	5	∞	1	1	1	0
	2	∞	0	-2	1	0	2	2	2
	3	4	**7**	0	∞	3	**1**	3	0
	4	∞	∞	-6	0	0	0	4	4

		d				**P**			
		1	2	3	4	1	2	3	4
$k=2$	1	0	3	**1**	**4**	1	1	**2**	**2**
	2	∞	0	-2	1	0	2	2	2
	3	4	7	0	**8**	3	1	3	**2**
	4	∞	∞	-6	0	0	0	4	4

Primero copiamos fila y columna 2 (k=2) en ambas matrices. Para cada celda restante comprobamos si $d_{ij} > d_{i2}+d_{2j}$ si es así hacemos $d_{ij} = d_{i2}+d_{2j}$ y $P_{ij} = P_{2j}$. En otro caso, copiamos lo que había en la celda de la matriz anterior. Pero, esto se automatiza como indican las flechas para no enredarnos con los sub-índices.

A estas alturas, seguro ha quedado claro la automatización del algoritmo en papel. Continuamos sin ser tan explícitos en la explicación.

8.7 Algoritmo de Floyd-Warshall (C4). Traza.

$k=2$

	d				P			
	1	2	3	4	1	2	3	4
1	0	3	**1**	**4**	1	1	**2**	**2**
2	∞	0	-2	1	0	2	2	2
3	4	7	0	**8**	3	1	3	**2**
4	∞	∞	-6	0	0	0	4	4

$k=3$

	d				P			
	1	2	3	4	1	2	3	4
1	0	3	1	4	1	1	2	2
2	**2**	0	-2	1	**3**	2	2	2
3	4	7	0	8	3	1	3	2
4	**-2**	**1**	-6	0	**3**	**1**	4	4

$k=3$

	d				P			
	1	2	3	4	1	2	3	4
1	0	3	1	4	1	1	2	2
2	**2**	0	-2	1	**3**	2	2	2
3	4	7	0	8	3	1	3	2
4	**-2**	**1**	-6	0	**3**	**1**	4	4

$k=4$

	d				P			
	1	2	3	4	1	2	3	4
1	0	3	-2	4	1	1	**4**	2
2	-1	0	-5	1	**3**	2	**4**	2
3	4	7	0	8	3	1	3	2
4	-2	1	-6	0	3	1	4	4

8.7 Algoritmo de Floyd-Warshall (C4). Traza.

Identificación de los árboles de caminos mínimos.

Para ello en la matriz *P* óptima, cada fila representa el vector *pred* como se vio en los problemas tipo B donde *s es* el nodo índice de la fila. Para el ejemplo anterior.

Matrices Óptimas

	d				***P***			
	1	2	3	4	1	2	3	4
1	0	3	-2	4	1	1	4	2
2	-1	0	-5	1	3	2	4	2
3	4	7	0	8	3	1	3	2
4	-2	1	-6	0	3	1	4	4