Apuntes de Grafos

Un grafo es una entidad matemática introducida por Euler en 1736 para representar entidades (*vértices*) que pueden relacionarse libremente entre sí, mediante el concepto de *arista*. Se puede definir un TAD Grafo basado en estas ideas, el cual contiene elementos sobre los que esta definida una *relación de vecindad o adyacencia*. Un vértice puede relacionarse con cualquier otro vértice y establecer cualquier número de relaciones.

Hay muchas situaciones en las cuales el modelado más conveniente de los datos de una aplicación es mediante grafos, por ejemplo la representación de una red de carreteras, calles, telecomunicaciones, electrificación, internet, planificación de tareas, etapas de un proceso industrial, etc.

1. Definiciones

Un **grafo G** = (**V**, **A**) se define por:

- Un conjunto de *n* **vértices**, **V**, a los cuales se hace referencia por sus índices 1…*n*.
- Un conjunto de *m* **aristas**, **A**, que conectan vértices entre sí. Una arista es un par de vértices, indicados de la forma $\langle i, j \rangle$. Si $\langle i, j \rangle \in \mathbf{A}$ significa que el vértice *i* está conectado con el vértice *j*.
- No existen aristas de la forma $\langle i, i \rangle$ (que conecten un vértice consigo mismo).

Un **subgrafo de G** es cualquier grafo **G'** = (**V**, **A'**) donde **A'** sea un subconjunto de **A**

Dependiendo de si el orden de los vértices en las aristas importa o no tenemos:

- **Grafo dirigido**: El orden importa, $\langle i, j \rangle \neq \langle j, i \rangle$. El que el vértice *i* esté conectado con el vértice *j* no implica que el vértice *j* esté conectado con el vértice *i*.
- **Grafo no dirigido**: El orden no importa, $\langle i, j \rangle \equiv \langle j, i \rangle$. $\langle i, j \rangle \in \mathbf{A} \Leftrightarrow \langle j, i \rangle \in \mathbf{A}$.

Ejemplos de grafos y su representación:

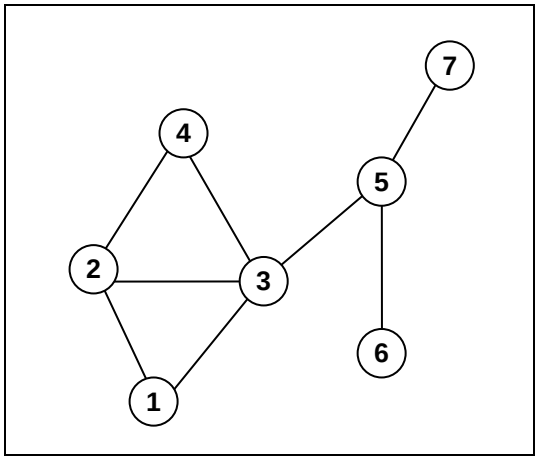


Fig. 1: Grafo no dirigido

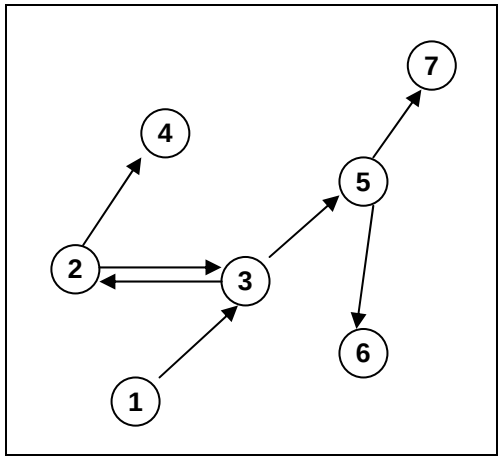


Fig.2: Grafo dirigido

El grafo de la figura 1 se define por: **V** = [1..7], **A** = { $\langle 1,2 \rangle$, $\langle 1,3 \rangle$, $\langle 2,1 \rangle$, $\langle 2,3 \rangle$, $\langle 2,4 \rangle$, $\langle 3,1 \rangle$, $\langle 3,2 \rangle$, $\langle 3,4 \rangle$, $\langle 3,5 \rangle$, $\langle 4,2 \rangle$, $\langle 4,3 \rangle$, $\langle 5,3 \rangle$, $\langle 5,6 \rangle$, $\langle 5,7 \rangle$, $\langle 6,5 \rangle$, $\langle 7,5 \rangle$ }.

El grafo de la figura 2 se define por: **V** = [1..7], **A** = { $\langle 1,3 \rangle$, $\langle 2,3 \rangle$, $\langle 2,4 \rangle$, $\langle 3,2 \rangle$, $\langle 3,5 \rangle$, $\langle 5,6 \rangle$, $\langle 5,7 \rangle$ }.

En muchas aplicaciones de los grafos las aristas llevan asociada información adicional. En ese caso hablaremos de **grafos etiquetados**. Si esa información es numérica y tiene el significado del *coste* necesario para recorrer esa arista, entonces usaremos el nombre de **grafo ponderado** o **red**.

Red: Grafo en el que cada arista lleva asociado un **coste** (de aquí en adelante lo llamaremos **longitud**)

Definiremos la función **longitud** entre los vértices *i* y *j* de una red como:

$$long(i,j) = \begin{cases} 0 & \text{si } i = j \\ \infty & \text{si } \langle i, j \rangle \notin \text{A} \\ \text{coste}(\langle i, j \rangle) & \text{si } \langle i, j \rangle \in \text{A} \end{cases}$$

Para un grafo que no sea una red se supone que todas las aristas tienen coste unidad.

2. Terminología

- Dos vértices *i* y *j* son **adyacentes** si existe una arista que los conecte (es decir, si $\langle i, j \rangle \in \mathbf{A}$)
- El **grado** de un vértice en un grafo no dirigido es igual al número de vértices adyacentes a él (número de aristas donde el vértice aparece como el primer o segundo componente de la arista). Por ejemplo, en la figura 1 el vértice 1 tiene grado 2, y el vértice 3 tiene grado 4.
- En un grafo dirigido se distingue entre **grado interior** de un vértice (número de aristas que llegan a él, es decir aristas donde el vértice aparece como segundo componente) y **grado exterior** (número de aristas que salen de él, aristas donde el vértice aparece como primer componente). Por ejemplo en la figura 2 el vértice 1 tiene grado interior 0 y grado exterior 1, y el vértice 3 tiene grado interior 2 y grado exterior 2.
- Un **camino** entre dos vértices *i* y *j* es cualquier secuencia de vértices, $v_1, \ldots, v_k, \ldots, v_p$ que cumpla que $v_1 = i$, $v_p = j$ y exista una arista entre cada par de vértices contiguos: $\forall\, k : \langle v_k, v_{k+1} \rangle \in \mathbf{A}$
- Por ejemplo, en la figura 1, los siguientes serían caminos posibles entre los vértices 1 y 5:
  - 1,3,5
  - 1,3,4,2,3,5
  - 1,3,4,2,3,4,2,3,5
- Un **camino simple** es aquel donde no hay vértices repetidos en la secuencia, salvo el primero y el último (que pueden ser iguales o distintos). Un **ciclo** es un camino simple donde el vértice inicial y el final son el mismo ($i = j$). Por ejemplo, 1,3,5 sería un camino simple, 1,3,2,1 también (y además un ciclo), pero 1,3,4,2,3,5 no es camino simple.
- Un grafo se dice que es **acíclico** si todos sus posibles caminos son simples (no existen ciclos). Por ejemplo el grafo de la figura 1 sería acíclico si eliminásemos los vértices ⟨1,2⟩ y ⟨2,4⟩ (se entiende que automáticamente desaparecen ⟨2,1⟩ y ⟨4,2⟩).
- Un **árbol** es un grafo **no dirigido** y **acíclico**.
- Se dice que un grafo está **conectado** si existe como mínimo un camino entre cualquier par de vértices distintos. Por ejemplo, el grafo de la figura 1 está conectado, pero el de la figura 2 no (no hay camino que vaya de 3 a 1, etc.)
- Un grafo esta **completamente conectado** si para cada vértice existen aristas que lo conecten con los n-1 vértices restantes. Este tipo de grafo tiene el número máximo posible de aristas, $n \cdot (n-1)/2$. Por lo tanto $m \in O(n^2)$
- Si un grafo contiene ciclos, el número de posibles caminos es infinito. Si queremos enumerar el número de posibles caminos simples, el peor caso es un grafo completamente conectado, y entonces el número de posibles caminos simples es de $n!$

- Se define **longitud/coste de un camino** como suma de las longitudes de las aristas que recorre el camino (recordar que en un grafo que no sea red las aristas tienen longitud unidad). En la figura 1 la longitud del camino 1,3,5 sería 2.
- Se puede definir entonces la longitud de un camino por medio de la función longitud entre dos vértices, definida anteriormente: (no confundir ambas funciones, aunque tengan el mismo nombre sus parámetros son distintos)

$$long\left(\left[v_1,\ldots,v_p\right]\right)=\sum_{k=1}^{p-1} long(k,k+1)$$

- Para un grafo conectado, se define **ruta optima** entre los vértices *i* y *j* como el camino de longitud mínima entre los vértices *i* y *j*.
- Se denominará **distancia** entre los vértices *i* y *j* a la longitud de su ruta óptima.

3. Representaciones de grafos

Aunque en general al representar datos de la vida real mediante un grafo los vértices suelen llevar asociada información, en lo que sigue supondremos que esa información se almacena en una lista indexada y por lo tanto podemos hacer referencia a los vértices utilizando únicamente el índice donde están almacenados en esa lista. En lo que sigue las representaciones hacen referencia únicamente a la manera de almacenar las aristas.

Las operaciones básicas sobre grafos son las de comprobación de existencia de arista entre dos vértices (o conocer su longitud, si el grafo es etiquetado), recorrer la lista de vértices adyacentes a uno dado, la inserción y borrado de una arista, y la inserción y borrado (junto con las aristas asociadas) de un vértice. En muchas aplicaciones, sin embargo, el conjunto de vértices no varía durante la ejecución.

Las dos representaciones principales de grafos son las siguientes:

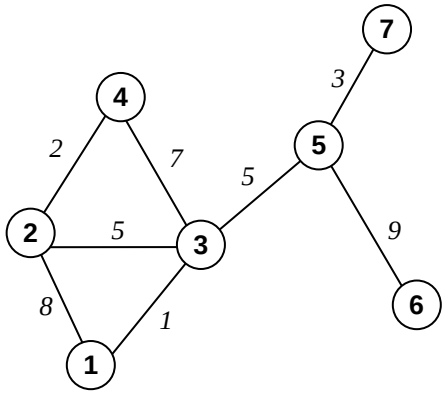
- **Matriz de Adyacencia** (**MA**): Se utiliza una matriz de tamaño $n \times n$ donde las filas y las columnas hacen referencia a los vértices para almacenar en cada casilla la longitud entre cada par de vértices del grafo. La celda **MA**[*i*, *j*] almacena la longitud entre el vértice *i* y el vértice *j*. Si su valor es infinito significa que no existe arista entre esos vértices, y MA[*i*, *i*] = 0.
- **Lista de Adyacencia** (**LA**): Se utiliza un vector de tamaño *n* (un elemento por cada vértice) donde **LA**[*i*] almacena la referencia a una lista de los vértices adyacentes a *i*. En una red esta lista almacenará también la longitud de la arista que va desde *i* al vértice adyacente.

  Existen varias posibilidades a la hora de representar la lista de vértices: arrays dinámicos, listas enlazadas o usar una **lista de adyacencia aplanada**: Se almacenan todas las listas de manera contigua en un único vector, **VA**, de tamaño *m*, y en el vector **LA** se almacenan índices al vector **VA**. La lista de adyacencia del vértice *i* se encuentra en **VA[LA**[*i*] .. **LA**[*i*+1]-1]. Esta representación es útil cuando no se vaya a modificar el grafo.

La eficiencia de las operaciones básicas en cada representación es la siguiente:

Operación	Matriz de Adyacencia	Lista de Adyacencia
Espacio ocupado	$\Theta(n^2)$	$\Theta(m+n)$
Longitud entre vértices / Existencia de aristas	$O(1)$	$O(\text{grado}(i))$
Recorrido vértices adyacentes a *i*	$\Theta(n)$	$\Theta(\text{grado}(i))$
Recorrido todas las aristas	$\Theta(n^2)$	$\Theta(m)$
Inserción/borrado arista	$O(1)$	$O(\text{grado}(i))$
Inserción/borrado vértice	$O(n^2)$	$O(n)$

Ejemplo. Dado la siguiente red no dirigida, donde se indican la longitud/coste de cada arista:



Representación mediante matriz de adyacencia:

vértice \ vértice	1	2	3	4	5	6	7
1	0	8	1	∞	∞	∞	∞
2	8	0	5	2	∞	∞	∞
3	1	5	0	7	5	∞	∞
4	∞	2	7	0	∞	∞	∞
5	∞	∞	5	∞	0	9	3
6	∞	∞	∞	∞	9	0	∞
7	∞	∞	∞	∞	3	∞	0

Representación mediante listas de adyacencia: En las listas se almacenan pares (**vértice adyacente**, longitud de arista).

vértice					
1	(**2**,8)	(**3**,1)			
2	(**1**,8)	(**3**,5)	(**4**,2)		
3	(**1**,1)	(**4**,7)	(**2**,5)	(**5**,5)	
4	(**2**,2)	(**3**,7)			
5	(**3**,5)	(**7**,3)	(**6**,9)		
6	(**5**,9)				
7	(**5**,3)				

Representación mediante listas de adyacencia aplanadas (vector de índices, **LA**, y de adyacencia, **VA**):

1	**2**	**3**	**4**	**5**	**6**	**7**
1	3	6	10	12	15	16

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
(**2**,8)	(**3**,1)	(**1**,8)	(**3**,5)	(**4**,2)	(**1**,1)	(**4**,7)	(**2**,5)	(**5**,5)	(**2**,2)	(**3**,7)	(**3**,5)	(**7**,3)	(**6**,9)	(**5**,9)	(**5**,3)

4. Problema de la ruta óptima (shortest path)

Objetivo: Dada una red conectada, encontrar la ruta óptima (la de menor distancia) entre los vértices *i* y *j*.

Representación de la solución: La solución consistirá en proporcionar la secuencia de vértices, $v_1 \ldots v_k \ldots v_p$ que forman la ruta óptima ($v_1 = i$, $v_p = j$)

4a) Algoritmo Backtracking / Fuerza Bruta

El espacio de posibles soluciones son todos los posibles caminos simples que van desde el vértice *i* al vértice *j*. Una solución fuerza bruta consistiría en generar todos esos posibles caminos, calcular su coste (longitud), y obtener el mínimo.

Para generar todos los posibles caminos simples lo más sencillo es utilizar un enfoque recursivo, basado en las ideas de la técnica de backtracking: Para ir del vértice *i* hasta el *j* se debe primero ir desde *i* hasta uno de sus vértices adyacentes, *k* (que puede ser el propio *j*). Para generar todos los caminos basta entonces con un bucle en el que se recorren todos los vértices adyacentes de *i* y para cada uno de ellos (vértice *k*) se pide, recursivamente, obtener todos los caminos desde *k* hasta *j*.

Como queremos los caminos simples, y en ellos no se debe pasar por cada vértice más de una vez, iremos marcando cada vértice ya visitado y no tomaremos en cuenta esos vértices si les encontramos en el proceso de generar un camino.

Esquema de algoritmo:

```
ruta_optima(i, j, ruta, ruta_optima)
{ Calcula la ruta óptima entre i y j y la concatena en la lista ruta_optima. }
    si i = j entonces
        medir_ruta(ruta)
        si es mejor que ruta_optima entonces ruta_optima ← ruta
    sino
        marcar i como visitado
        ∀ k : k no visitado, k adyacente a i :
        ⌈   añadir k al final de ruta
        |   ruta_optima(k, j, ruta, ruta_optima)
        ⌊   quitar k del final de ruta
        marcar i como no visitado
    fin
```

Mejoras del algoritmo: Se puede ir calculando la longitud de la ruta a medida que se va seleccionando *k*, se puede evitar el insertar y borrar cada valor de *k* almacenando únicamente el *k* óptimo en el regreso de las llamadas recursivas, se puede poner como criterio de poda que si el subcamino generado hasta el momento ya tiene un coste/longitud mayor que el mejor encontrado entonces no merece la pena considerarle.

Eficiencia: El número de operaciones dependerá del número de posibles caminos simples que haya en el grafo. En el peor caso, para un grafo completamente conectado, desde el primer vértice podemos pasar a cualquiera de los *n*-1 vecinos, desde ese vecino tenemos acceso a *n*-2 vértices (quitamos el primero), desde ese a *n*-3, y así sucesivamente. El número de posibles caminos será de $O(n!)$

Para un grafo donde cada vértice tenga un grado *g* (*g* vértices adyacentes), siguiendo el razonamiento anterior suponiendo que no encontramos vértices ya visitados obtenemos una cota superior de $O((g\text{-}1)^n)$**.**

En cualquier caso, salvo para grafos muy poco conectados (sólo un vecino) el tiempo es **no polinómico**.

4b) Programación Dinámica – Algoritmo de Floyd

Para aplicar programación dinámica al problema anterior primero resolveremos el **problema restringido**: En este caso (problema de optimización) lo normal es pedir la longitud de la ruta óptima entre *i* y *j* (lo que llamamos *distancia* entre *i* y *j*) y renunciar a saber cuál es esa ruta optima.

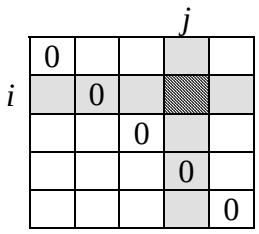
Es conveniente (no imprescindible, aunque simplifica el análisis) generalizar un poco el esquema backtracking visto anteriormente y hacer que *k* no tenga que ser un vértice adyacente a *i* sino que pueda ser **cualquier vértice intermedio** del camino que va desde *i* a *j* (puede ser el propio *j*, aunque debe ser distinto de *i*).

El problema de calcular la función *distancia* entre *i* y *j* (recordar que es la longitud de la **ruta óptima** entre *i* y *j*) se plantea entonces recursivamente como el problema de calcular el mínimo de la distancia entre *i* y *k* más la distancia entre *k* y *j* para todo posible *k*:

$$dist(i,j) = \begin{cases} 0 & \text{si } i = j \\ \min\limits_{\forall k \neq i} \{dist(i,k) + dist(k,j)\} & \text{si } i \neq j \end{cases}$$

En este caso comprobar el principio de suboptimalidad es trivial: Se puede ver fácilmente que entre todas las rutas entre *i* y *k* (y entre *k* y *j*) tan sólo son relevantes para la solución aquellas que sean óptimas, las de menor longitud. No es posible obtener una ruta óptima mediante subrutas no óptimas.

Al intentar aplicar la técnica de la tabla de resultados parciales (sustituyendo la función *dist*(*i*, *j*) por una matriz, *dist*[*i*, *j*]) nos encontramos con un problema insalvable: Es imposible encontrar una manera de rellenar la matriz en la que tengamos ya asignadas las celdas que necesitamos:



Para asignar la celda rayada necesitaríamos tener asignadas las celdas sombreadas.

Para resolver éste problema debemos (como es habitual en programación dinámica) generalizar la función que queremos calcular. Floyd encontró la siguiente generalización:

- *dist*(*i*, *j*, *k*) ≡ longitud de la ruta óptima entre *i* y *j* **donde los posibles vértices intermedios** son únicamente los vértices 1…*k*.
- *dist*(*i*, *j*, *n*) es la distancia entre *i* y *j* (todos los vértices están disponibles) y por lo tanto el valor que queremos calcular.
- *dist*(*i*, *j*, 0) = *long*(*i*, *j*). Cuando no existe ningún vértice intermedio disponible la única posibilidad de ir de *i* a *j* es que exista una arista que los conecte.

El objetivo es poder calcular *dist*(*i*, *j*, *k*) en función de *dist*(·, ·, *k*-1): La idea clave es darse cuenta que la ruta óptima entre *i* y *j* con puntos intermedios 1…*k* o bien tiene como punto intermedio a *k* o bien no lo tiene. En el segundo caso, *k* no interviene en la solución y por lo tanto será igual a la ruta óptima sin *k* como punto intermedio, y en el primer caso la solución será la ruta óptima desde *i* hasta *k* junto con la ruta óptima desde *k* hasta *j*, y en esas subrutas *k* **no es un punto intermedio** (es punto inicial o final).

Por lo tanto basta examinar cual de los dos casos es el óptimo. El cálculo queda así:

$$dist(i,j,k) = \begin{cases} long(i,j) & \text{si } k = 0 \\ \min\{dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1)\} & \text{si } k > 0 \end{cases}$$

Ahora podemos aplicar la técnica de la tabla de resultados parciales sin problemas. La matriz es tridimensional y sólo necesitamos asignarla en orden creciente de *k*. Como sólo necesitamos la submatriz *k*-1 en cada paso, realmente con una matriz bidimensional nos basta.

Como es habitual en programación dinámica, no sólo obtenemos la solución del problema sino también las soluciones de otros problemas relacionados. En este caso, además de la ruta óptima entre *i* y *j* obtenemos **todas las rutas óptimas entre todos los vértices del grafo**.

Para recuperar la solución general basta con almacenar la información de cuál es un punto intermedio cualquiera (*k*) por el que pasa la ruta óptima entre *i* y *j*. Como sabemos las rutas entre todos los vértices, bastara preguntar (recursivamente) por la ruta óptima entre *i* y *k* y entre *k* y *j*. Cuando no exista punto intermedio entonces debe existir una arista que conecte *i* y *j* y la ruta óptima es esa arista.

Declaramos las siguientes matrices que almacenarán la solución del problema:

- **D**[*i*, *j*], de tamaño $n \times n$, almacena en cada celda la distancia entre *i* y *j* (longitud de la ruta óptima entre *i* y *j*)
- **P**[*i*, *j*], de tamaño $n \times n$, almacena en cada celda el índice de un vértice intermedio de la ruta óptima que une *i* y *j*, o el valor -1 si no existe punto intermedio (ruta directa por arista que une *i* y *j*)

El esquema del algoritmo sería:

```
∀ i, j : D[i, j] ← long(i, j)
∀ i, j : P[i, j] ← −1
∀ k = 1..n :
    ∀ i, j :
        si D[i, j] > D[i, k]+D[k, j] entonces
            D[i, j] ← D[i, k]+D[k, j]
            P[i, j] ← k
        fin
```

El algoritmo es extremadamente elegante: 3 bucles anidados con un cálculo del valor mínimo en el bucle más interno. Para escribir la ruta entre *i* y *j* basta con llamar a la siguiente función:

```
escribe_ruta(i, j)
    si P[i, j] = −1 entonces
        escribir("<"; i ; "," ; j ; ">")
    sino
        escribe_ruta(i, P[i, j])
        escribe_ruta(P[i, j], j)
    fin
```

Eficiencia del algoritmo: El cálculo es trivial. El tiempo es $\Theta(n^3)$ y el espacio es $\Theta(n^2)$.

Generalizaciones: Este algoritmo se puede adaptar a otros problemas, entre ellos:

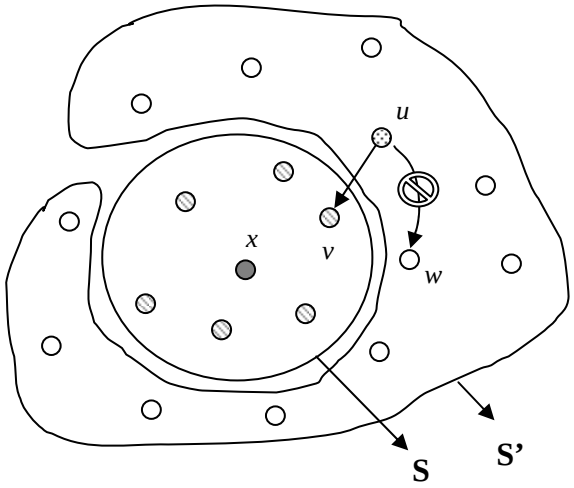
- Comprobación del cierre transitivo de un grafo (algoritmo de Warshall)
- Inversión de matrices reales (algoritmo de Gauss-Jordan)
- Cálculo del flujo máximo

4c) Algoritmo Voraz – Algoritmo de Dijkstra

El algoritmo de Dijkstra calcula la ruta óptima desde un vértice (*x*, vértice origen) **a todos los demás** vértices del grafo. Tiene como requisito que **no existan longitudes/costes negativos**. Se basa en la idea de la **relajación de aristas** (edge relaxation), en la cual se mantiene información de cual es la mejor distancia conocida de cualquier vértice al origen, y a medida que se va conociendo una distancia mejor desde el vértice *j* al origen, se actualizan las mejores distancias conocidas a todos los vértices adyacentes a *j*.

En su desarrollo utiliza un enfoque **voraz**: En un momento dado hay un conjunto de vértices, que llamaremos **S**, de los cuales se conoce la ruta óptima al origen y además son los **más cercanos** a él (cuando se hallen las rutas óptimas del resto de vértices ninguna tendrá una longitud menor que las de los vértices que están en **S**). Del resto de los vértices (conjunto **S'**) no conocemos la ruta óptima, sino la mejor ruta encontrada hasta el momento (no tiene porqué ser la óptima).

Se puede visualizar el algoritmo imaginando que en cada momento tenemos un círculo centrado en el origen que contiene los vértices más cercanos. En cada paso ampliamos el círculo incluyendo el vértice de fuera más cercano (ampliamos por círculos concéntricos).



En cada paso se halla el vértice de **S'** a menor distancia del origen y se obtiene su ruta óptima. Ese vértice se elimina de **S'** y pasa a formar parte de **S**. En este punto aplicamos la relajación de aristas y actualizamos las mejores rutas de los vértices que quedan en **S**.

La clave para que éste algoritmo obtenga la solución óptima reside en que el vértice (llamémosle *u*) de **S'** que se debe incorporar a **S** tiene que ser uno para el cual su ruta óptima vaya directamente de *u* a un vértice de **S** (llamémosle *v*), y su ruta óptima no puede ser una donde se vaya de *u* a un vértice $w \in$ **S'** ya que entonces (al no existir longitudes negativas), la ruta de *u* al origen sería más larga que la de *w* al origen, y entonces sería *w* el vértice elegido para incorporarse a **S**, y no *u*.

Si se han realizado relajaciones de aristas de los vértices de **S**, entonces ya se conoce la ruta óptima de vértices de **S'** al origen pasando directamente por vértices de **S** (**Nota**: Esto no es lo mismo que conocer la ruta óptima de vértices de **S'** al origen, la condición de que se pase directamente por vértices de **S** puede ser excesiva), y el vértice cuya ruta de esas características sea la más corta será aquel que podemos incorporar a **S** porque podemos estar seguros de que esa ruta es óptima.

En cualquier etapa del algoritmo un vértice o bien pertenece a **S** o bien a **S'**, por lo que $\mathbf{S} \cup \mathbf{S'} = [1..n]$. Inicialmente **S** estará vacío y **S'** = [1..*n*], y el primer vértice que se incorporará a **S** será *x* (el origen).

En el esquema del algoritmo vamos a definir dos vectores, ambos de tamaño *n* que van a almacenar la solución:

- **D**[*i*] almacenará la distancia (longitud de la ruta óptima) de *x* a *i*, si $i \in$ **S**, o la longitud de la mejor ruta conocida si $i \in$ **S'**.
- **P**[*i*] almacenará el *predecesor* (vértice anterior en la ruta) de *i* en la ruta óptima de *x* a *i*, si $i \in$ **S**, o el predecesor en la mejor ruta conocida si $i \in$ **S'**.

El esquema del algoritmo sería (se marcan con números operaciones que se analizarán posteriormente):

```
{ inicialización }
S' ← [1..n]
∀ i : D[i] ← ∞, P[i] ← -1
D[x] ← 0
{ iteración voraz }
repetir n veces
    { encontrar vértice que se incorpora a S }
    ① u ← min_{u∈S'} {D[u]}
    ② excluir u de S'
    { relajación de vértices de S' adyacentes a u }
    ③ ∀ k : (k ∈ S') ∧ (k adyacente a u) :
        si D[k] > D[u] + long(u, k) entonces
            { existe ruta mejor de k a x pasando de k a u y de u a x }
            ④ D[k] ← D[u] + long(u, k)
            P[k] ← u
        fin
fin
```

El resultado es que el vector **D**[*i*] nos indica la distancia entre *i* y cualquier otro vértice, y el vector **P** nos sirve para hallar la ruta óptima usando la siguiente función, parecida a la del algoritmo de Floyd:

```
escribe_ruta(i)
{ escribe la ruta óptima desde x a i }
    si i = x entonces
        escribe(x ," ")
    sino
        escribe_ruta(P[i]) { ruta desde x al predecesor de i }
        escribe(i , " ")
    fin
```

Análisis del algoritmo: El análisis dependerá de la representación elegida para el grafo y para el vector **D** (la representación óptima del conjunto **S'** es mediante un vector de *n* elementos booleanos donde cada celda indique si el vértice *i* pertenece o no al conjunto):

Dependiendo de si el grafo se representa por matriz o por lista de adyacencia tenemos:

- Matriz de adyacencia: El bucle ③ se itera *n* veces. En el total del algoritmo se itera n^2 veces.
- Lista de adyacencia: El bucle ③ se itera *grado*(*u*) veces. Como está dentro de un bucle en el que se seleccionan todos los vértices, en el total del algoritmo se itera *m* veces.

El vector **D** se puede representar como un vector, o bien, fijándonos en que tenemos que realizar la operación de extraer el mínimo, como un **montículo binario** que contenga únicamente vértices de **S'**.

- Vector: La operación ① es $O(n)$, las operaciones ② y ④ son $O(1)$.
- Montículo: La operación ① es $O(1)$, la operación ② es $O(\log n)$ ya que no solo hay que excluir del conjunto sino **eliminar** *u* del montículo, y la operación ④ es de $O(\log n)$ ya que al cambiar el valor de **D**[*k*] hay que **modificar** (descenso de clave) un elemento del montículo.

Por lo tanto tenemos dos alternativas principales:

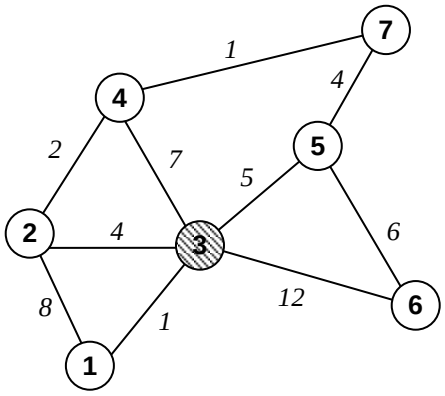
- Grafo por matriz/lista de adyacencia, **D** vector: $\Theta(n^2)$
- Grafo por lista de adyacencia, **D** montículo: $O((n+m)\cdot\log n)$. Como habitualmente $m > n$ podemos simplificar a $O(m \log n)$

Si el grafo está muy conectado entonces $m \in \Theta(n^2)$ y la primera alternativa es preferible. Sin embargo, para grafos que no estén muy conectados la segunda alternativa podría ser preferible.

Nota: Existen alternativas más eficientes, utilizando montículos de Fibonacci (no los contemplamos en la asignatura).

Ejemplo de la ejecución del algoritmo:

El origen es $x = 3$. Se muestran sombreadas las celdas de los vértices que pertenecen a **S** (y por lo tanto no se actualizan) y en negrilla los cambios (por relajación de aristas) que se producen:



Iteración	*u*	Vectores							S
			1	**2**	**3**	**4**	**5**	**6**	**7**	
Inicial	---	**D:**	∞	∞	0	∞	∞	∞	∞	[ ]
		P:	-	-	-	-	-	-	-	
1	3	**D:**	1	4	0	7	5	12	∞	[3]
		P:	3	3	-	3	3	3	-	
2	1	**D:**	1	4	0	7	5	12	∞	[1,3]
		P:	3	3	-	3	3	3	-	
3	2	**D:**	1	4	0	**6**	5	12	∞	[1..3]
		P:	3	3	-	**2**	3	3	-	
4	5	**D:**	1	4	0	6	5	**11**	**9**	[1..3,5]
		P:	3	3	-	2	3	**5**	**5**	
5	4	**D:**	1	4	0	6	5	11	**8**	[1..5]
		P:	3	3	-	2	3	5	**4**	
6	7	**D:**	1	4	0	6	5	11	8	[1..5,7]
		P:	3	3	-	2	3	5	4	
7	6	**D:**	1	4	0	6	5	11	8	[1..7]
		P:	3	3	-	2	3	5	4	