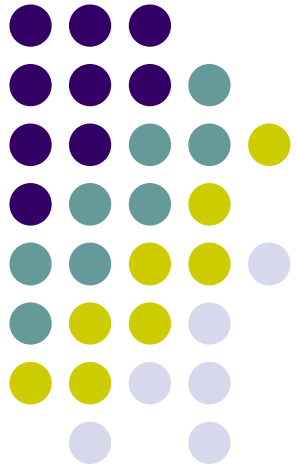
Algoritmos de Grafos



Departamento de Informática
Universidad de Valladolid

César González Ferreras

Contenidos

- Introducción
- Representación de grafos
- Camino mínimo
- Búsqueda en grafos
- Árbol de recubrimiento mínimo
- Ordenación topológica

INTRODUCCIÓN

Introducción

- Un grafo es una entidad matemática introducida por Euler en 1736
- Permite representar:
  - Entidades: **vértices**
  - Relaciones entre vértices: **aristas**
- Un grafo establece 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:
  - Red de carreteras, calles
  - Red de telecomunicaciones, electrificación, internet
  - Planificación de tareas, etapas de un proceso industrial

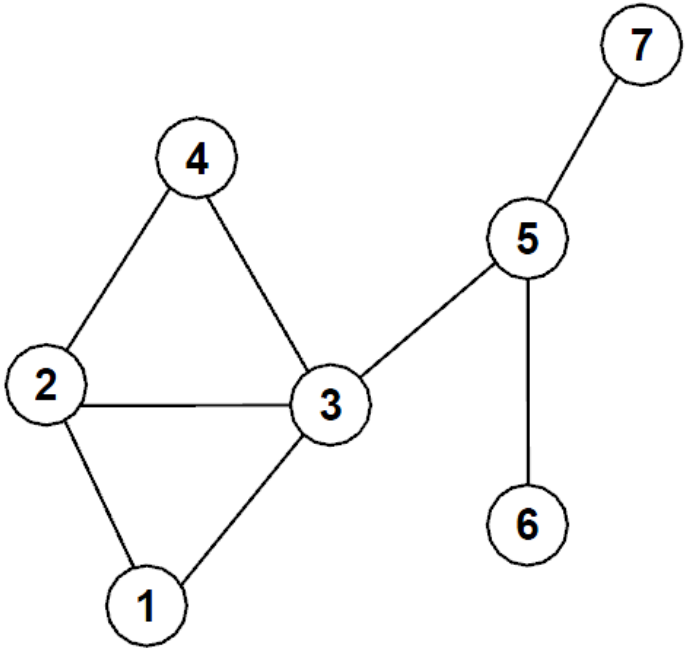
Definiciones

- 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 $\langle i, j \rangle \in$ **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**

Definiciones

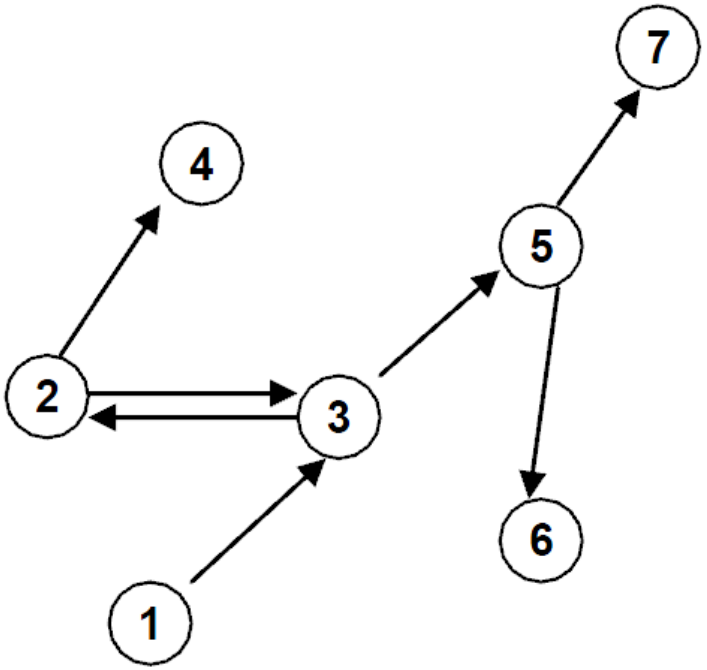
- Dependiendo de si el orden de los vértices en las aristas importa o no tenemos:
  - **Grafo dirigido**:
    - El orden importa, $\langle i, j \rangle \neq \langle j, i \rangle$. El que el vértice *i* esté conectado con el vértice *j* no implica que el vértice *j* esté conectado con el vértice *i*.
  - **Grafo no dirigido**:
    - El orden no importa, $\langle i, j \rangle \equiv \langle j, i \rangle$. $\langle i, j \rangle \in \mathbf{A} \Leftrightarrow \langle j, i \rangle \in \mathbf{A}$.

Ejemplo de grafo no dirigido



- **V** = [1..7]
- **A** = { ⟨1,2⟩ , ⟨1,3⟩ , ⟨2,1⟩ , ⟨2,3⟩ , ⟨2,4⟩ , ⟨3,1⟩ , ⟨3,2⟩ , ⟨3,4⟩ , ⟨3,5⟩ , ⟨4,2⟩ , ⟨4,3⟩ , ⟨5,3⟩ , ⟨5,6⟩ , ⟨5,7⟩ , ⟨6,5⟩ , ⟨7,5⟩ }.

Ejemplo de grafo dirigido



- **V** = [1..7]
- **A** = { ⟨1,3⟩ , ⟨2,3⟩ , ⟨2,4⟩ , ⟨3,2⟩ , ⟨3,5⟩ , ⟨5,6⟩ , ⟨5,7⟩ }.

Definiciones

- En muchas aplicaciones de los grafos las aristas llevan asociada información adicional:
  - **grafos etiquetados**
- Si esa información es numérica y tiene el significado del *coste* necesario para recorrer esa arista:
  - **grafo ponderado** o **red**.
- El **coste** se suele definir como una función que llamaremos **longitud** entre los vértices *i* y *j* :

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

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**:
  - 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 anterior 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
    - aristas donde el vértice aparece como segundo componente
  - **grado exterior:** número de aristas que salen de él
    - aristas donde el vértice aparece como primer componente
  - Por ejemplo en la figura anterior el vértice 1 tiene grado interior 0 y grado exterior 1, y el vértice 3 tiene grado interior 2 y grado exterior 2.

Terminología

- Un **camino** entre dos vértices *i* y *j* es cualquier secuencia de vértices, v_1 , … , v_k , … , v_p que cumpla que:
  - $v_1 = i$, $v_p = j$
  - exista una arista entre cada par de vértices contiguos: $\forall k$: $\langle v_k , v_{k+1} \rangle \in$ **A**
- Por ejemplo, en la figura anterior de grafo no dirigido, 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.

Terminología

- Un grafo se dice que es **acíclico** si todos sus posibles caminos son simples (no existen ciclos).
  - Por ejemplo el grafo no dirigido anterior sería acíclico si eliminásemos los vértices $\langle 1,2 \rangle$ y $\langle 2,4 \rangle$ (se entiende que automáticamente desaparecen $\langle 2,1 \rangle$ y $\langle 4,2 \rangle$).
- 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 no dirigido anterior está conectado, pero el de la figura 2 no (no hay camino que vaya de 3 a 1, etc.)

Terminología

- 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 $n!$

Terminología

- Se define **longitud/coste de un camino** como suma de las longitudes de las aristas que recorre el camino
- Se puede definir entonces la longitud de un camino por medio de la función longitud entre dos vértices, definida anteriormente:

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

Terminología

- Para un grafo conectado, se define **ruta óptima** 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

REPRESENTACIÓN DE GRAFOS

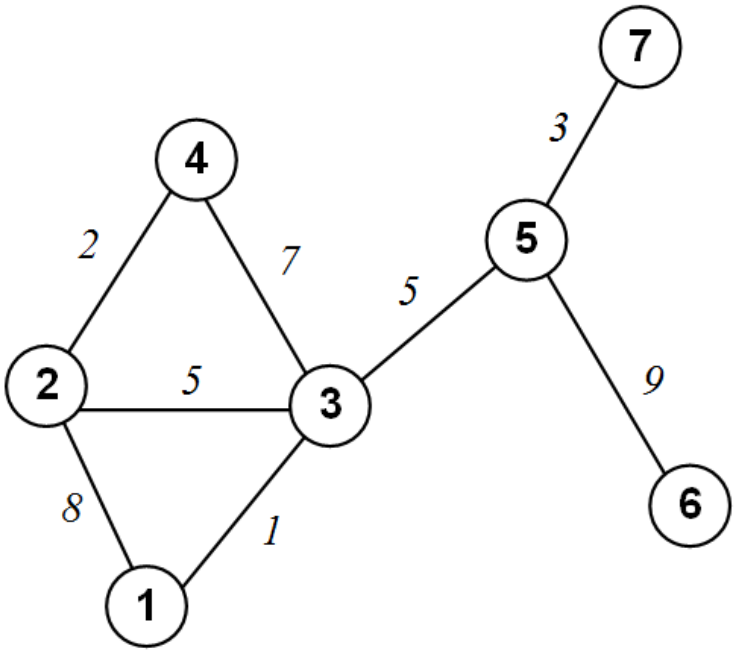
Representación de grafos

- Al representar datos de la vida real mediante un grafo los vértices suelen llevar asociada información:
  - esa información se almacena en una lista indexada.
- Las representaciones de grafos hacen referencia **únicamente a la manera de almacenar las aristas.**
- Las operaciones básicas sobre grafos son
  - Comprobación de existencia de arista entre dos vértices
    - Conocer su longitud si el grafo es etiquetado
  - Recorrer la lista de vértices adyacentes a uno dado
  - La inserción y borrado de una arista
  - La inserción y borrado (junto con las aristas asociadas) de un vértice.
- Las dos representaciones principales de grafos son las siguientes:
  - **Matriz de Adyacencia** (**MA)**
  - **Lista de Adyacencia** (**LA**)

Matriz de Adyacencia (MA)

- Se utiliza una matriz de tamaño $n \times n$
  - Las filas y las columnas hacen referencia a los vértices
- 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
  - MA[*i*, *i*] = 0.

Ejemplo matriz adyacencia

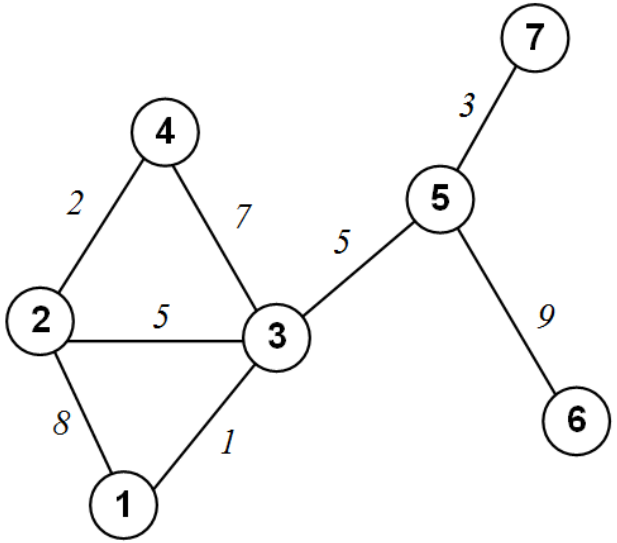


vértice → / vértice ↓	1	2	3	4	5	6	7
1	0	8	1	∞	∞	∞	∞
2	8	0	5	2	∞	∞	∞
3	1	5	0	7	5	∞	∞
4	∞	2	7	0	∞	∞	∞
5	∞	∞	5	∞	0	9	3
6	∞	∞	∞	∞	9	0	∞
7	∞	∞	∞	∞	3	∞	0

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. Lo más habitual:
  - arrays dinámicos
  - listas enlazadas

Ejemplo lista adyacencia



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)			

Eficiencia de las operaciones

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	$O(n^2)$	$O(m)$

- n – número de nodos
- m – número de aristas

CAMINO MÍNIMO

Cálculo de la ruta óptima

- **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 ($v_1 = i$, $v_p = j$)

Algoritmo Fuerza Bruta

- Una solución fuerza bruta consistiría en:
  - generar todos los posibles caminos simples que van desde el vértice *i* al vértice *j*.
  - calcular su coste (longitud),
  - obtener el mínimo.
- Lo más sencillo es utilizar un enfoque recursivo:
  - 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

Algoritmo Fuerza Bruta

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

Algoritmo Fuerza Bruta

- **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 $O((g-1)^n)$.**
  - En cualquier caso, salvo para grafos muy poco conectados (sólo un vecino) el tiempo es **no polinómico**

Algoritmo de Floyd

- Algoritmo de programación dinámica
- Se basa en generalizar el esquema de la solución de fuerza bruta 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*:

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

Algoritmo de Floyd

- Al intentar aplicar la técnica de la tabla de resultados parciales 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:
  - Hay una **dependencia circular de datos**
- Para resolver éste problema debemos **generalizar la función**
- 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.

Algoritmo de Floyd

- El objetivo es poder calcular *dist*(*i*, *j*, *k*) en función de *dist*(·, ·, *k*-1):
- Hay dos posibilidades a la hora de calcular la ruta óptima entre *i* y *j* con puntos intermedios 1…*k*:
  - La ruta n*o tiene como punto intermedio a k*
    - *k* no interviene en la solución y por lo tanto será igual a la ruta óptima sin *k* como punto intermedio
  - La ruta tiene como punto intermedio a *k*
    - 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

Algoritmo de Floyd

- La fórmula resultante es

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

- 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.
- Además de la ruta óptima entre *i* y *j* obtenemos **todas las rutas óptimas entre todos los vértices del grafo**.

Algoritmo de Floyd

- 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*)

Algoritmo de Floyd

$\forall\ i, j$: D[i, j] $\leftarrow$ long(i, j)

$\forall\ i, j$: P[i, j] $\leftarrow$ -1

$\forall\ k = 1..n$:

  $\forall\ i, j$:

    si D[i, j] > D[i, k]+D[k, j] **entonces**

      D[i, j] $\leftarrow$ D[i, k]+D[k, j]

      P[i, j] $\leftarrow$ k

    fin

Algoritmo de Floyd

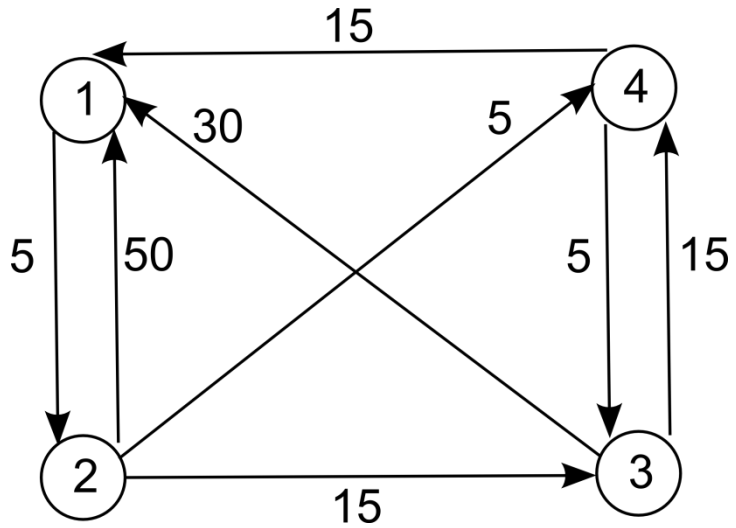
- Para escribir la ruta entre *i* y *j* basta con llamar a la siguiente función:

```
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
```

Algoritmo de Floyd

- **Eficiencia del algoritmo**:
  - El tiempo es $\Theta(n^3)$
  - El espacio es $\Theta(n^2)$.

Algoritmo de Floyd - Ejemplo



$$D_0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \qquad P_0 = \begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix}$$

Algoritmo de Floyd - Ejemplo

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{pmatrix} \qquad P_1 = \begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & \mathbf{1} & -1 & -1 \\ -1 & \mathbf{1} & -1 & -1 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & \mathbf{20} & \mathbf{10} \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} -1 & -1 & \mathbf{2} & \mathbf{2} \\ -1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ \mathbf{45} & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad P_3 = \begin{pmatrix} -1 & -1 & 2 & 2 \\ \mathbf{3} & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 \end{pmatrix}$$

Algoritmo de Floyd - Ejemplo

$$D_4 = \begin{pmatrix} 0 & 5 & \mathbf{15} & 10 \\ \mathbf{20} & 0 & \mathbf{10} & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad P_4 = \begin{pmatrix} -1 & -1 & \mathbf{4} & 2 \\ \mathbf{4} & -1 & \mathbf{4} & -1 \\ -1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 \end{pmatrix}$$

- Rutas:
  - 1-2
  - 1-2-4-3
  - 1-2-4
  - 2-4-1
  - 2-4-3
  - 2-4
  - 3-1
  - 3-1-2
  - 3-4
  - 4-1
  - 4-1-2
  - 4-3

Algoritmo de Dijkstra

- Es un algoritmo voraz
- 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*.

Algoritmo de Dijkstra

- 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.
- 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).

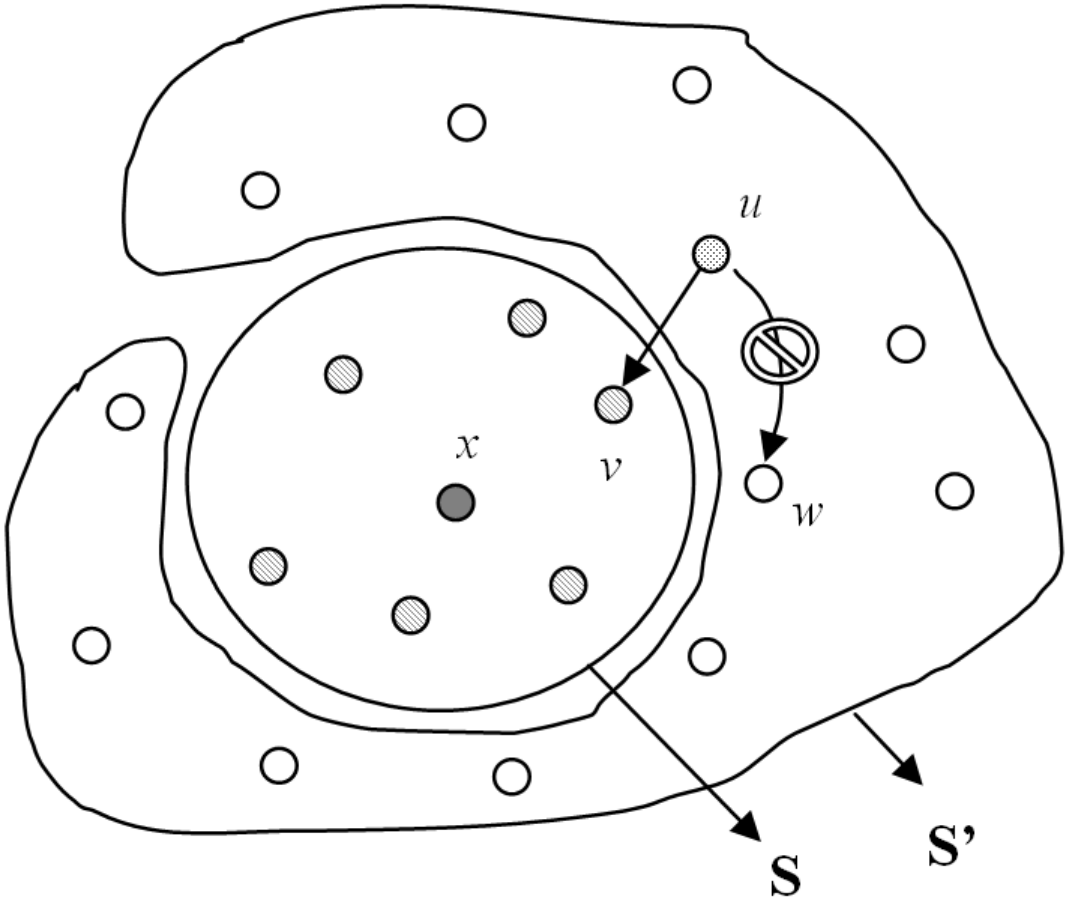
Algoritmo de Dijkstra

- 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’**.

Algoritmo de Dijkstra

- La clave para que este 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*.

Algoritmo de Dijkstra



Algoritmo de Dijkstra

- En cualquier etapa del algoritmo un vértice o bien pertenece a **S** o bien a **S'**
  - **S** $\cup$ **S'** = [1..*n*].
  - Inicialmente **S** estará vacío y **S'** = [1..*n*],
  - 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'**.

Algoritmo de Dijkstra

S' ← [1..*n*]

∀ *i* : D[*i*] ← ∞, P[*i*] ← -1

D[*x*] ← 0

repetir *n* **veces**

{ *encontrar vértice que se incorpora a S* }

①$u \leftarrow \min_{u \in S'} \{D[u]\}$

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

Algoritmo de Dijkstra

- El resultado es:
  - El vector **D**[*i*] nos indica la distancia entre *i* y cualquier otro vértice
  - El vector **P** nos sirve para hallar la ruta óptima usando la siguiente función:

```
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
```

Algoritmo de Dijkstra

- **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

Algoritmo de Dijkstra

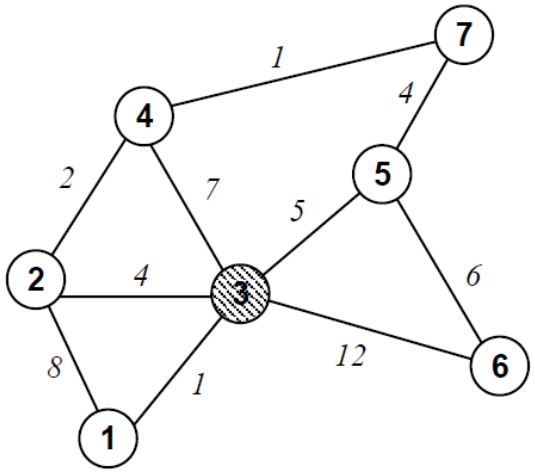
- El vector **D** se puede representar como un vector, o bien, fijándonos en que tenemos que realizar la operación de extraer el mínimo, como un **montículo binario** que contenga únicamente vértices de **S'**.
  - Vector: La operación ① es $O(n)$, las operaciones ② y ④ son $O(1)$.
  - Montículo: La 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.

Algoritmo de Dijkstra

- 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:**
  - $m \in \Theta(n^2)$ y la primera alternativa es preferible
- **Para grafos que no estén muy conectados:**
  - la segunda alternativa podría ser preferible.

Algoritmo de Dijkstra-Ejemplo

- El origen es *x* = 3. Se muestran sombreadas las celdas de los vértices que pertenecen a **S** y en negrilla los cambios 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	**7**	[1..5]
		P:	3	3	-	2	3	5	**4**	
6	7	**D:**	1	4	0	6	5	11	7	[1..5,7]
		P:	3	3	-	2	3	5	4	
7	6	**D:**	1	4	0	6	5	11	7	[1..7]
		P:	3	3	-	2	3	5	4	

- Rutas:
  - 3-1
  - 3-2
  - 3-2-4
  - 3-5
  - 3-5-6
  - 3-2-4-7

BÚSQUEDA EN GRAFOS

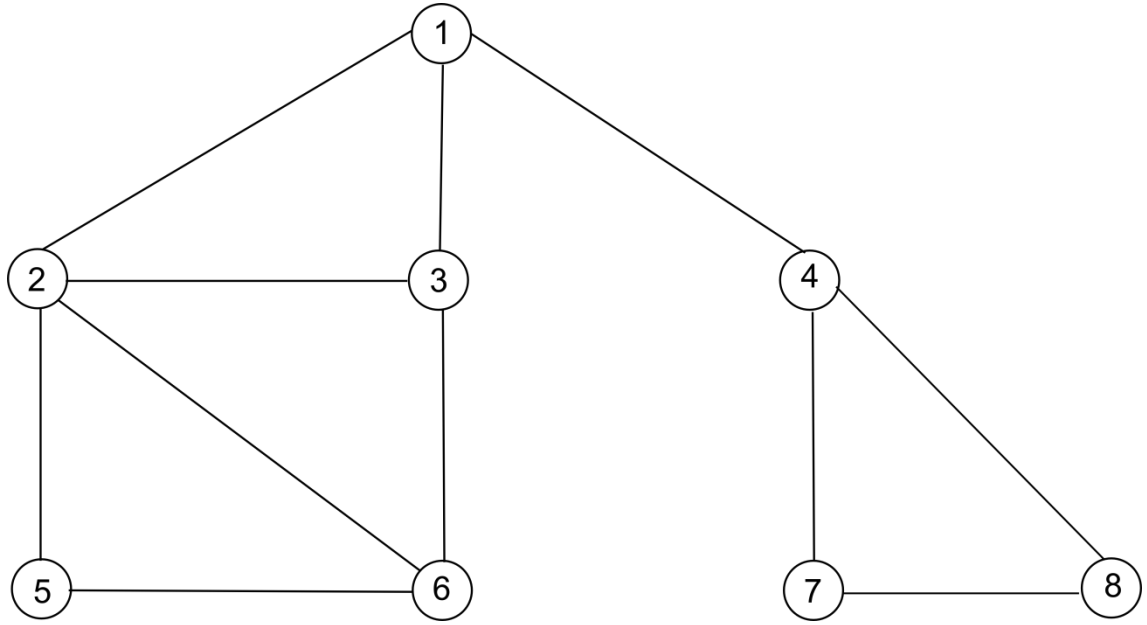
Recorrido en profundidad

- Un recorrido en profundidad (DFS o Depth First Search) es un algoritmo que permite recorrer todos los nodos de un grafo para realizar algún tipo de procesamiento sobre ellos.
- Sea G=<N,A> un grafo
- Es necesaria alguna manera de marcar un nodo para indicar que ya ha sido visitado.
- Funcionamiento:
  - Se selecciona un **nodo v** cualquiera y se marca como visitado.
  - Si hay un **nodo adyacente a v** que no ha sido visitado, se toma como punto de partida y se invoca recursivamente el mismo procedimiento.
  - Al retornar la llamada, si hay **otro nodo adyacente a v** que no haya sido visitado, se toma como punto de partida y se vuelve a llamar recursivamente.
  - Cuando **están marcados todos los nodos adyacentes a v**, retornamos.
  - **Si queda algún nodo en G por visitar**, se toma cualquiera de ellos como punto de partida y se vuelve a invocar la función recursiva.

Recorrido en profundidad

```
procedimiento recorridop(G)
    para cada v ϵ N hacer marca[v] ← no visitado
    para cada v ϵ N hacer
        si marca[v] ≠ visitado entonces rp(v)
procedimiento rp(v)
    marca[v] ← visitado
    para cada nodo w adyacente a v hacer
        si marca[w] ≠ visitado entonces rp(w)
```

Recorrido en profundidad - Ejemplo



- Si el recorrido empieza por el nodo 1 y suponemos que los nodos adyacentes se examinan en orden numérico, el recorrido en profundidad será:
  - 1, 2, 3, 6, 5, 4, 7, 8

Recorrido en profundidad - Coste

- Suponemos un grafo de n nodos y m aristas.
- Cada nodo se visita una vez, hay n llamadas al procedimiento rp()
  - Θ(n)
- Cuando visitamos un nodo, examinamos las marcas de los nodos vecinos.
  - Si usamos la representación Lista de Adyacencia, el coste total es:
    - Θ(m)
- El tiempo de ejecución será Θ(máx(n,m))

Versión no recursiva usando una pila

```
procedimiento recorridop(G)
    para cada v ϵ N hacer marca[v] ← no visitado
    para cada v ϵ N hacer
        si marca[v] ≠ visitado entonces rp2(v)
procedimiento rp2(v)
    P ← pila vacía
    marca[v] ← visitado
    apilar v en P
    mientras P no esté vacía hacer
        mientras exista un nodo w adyacente a cima(P) tal
                                        que marca[w] ] ≠ visitado hacer
            marca[w] ← visitado
            apilar w en P
        desapilar P
```

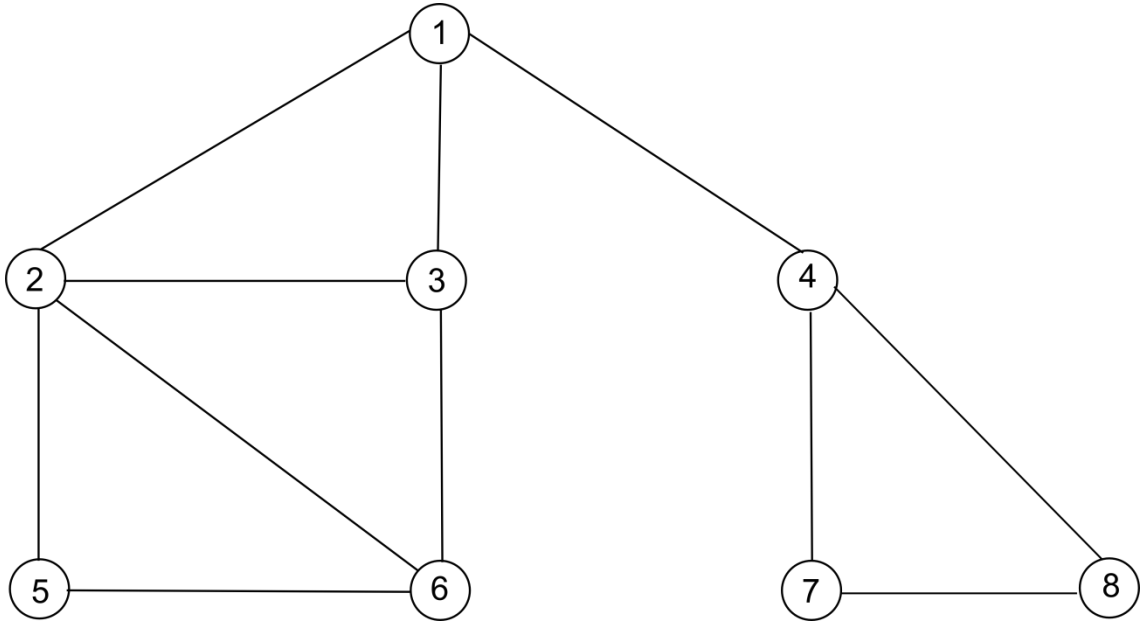
Recorrido en anchura

- El recorrido en anchura (BFS - Breadth First Search) es un algoritmo para recorrer todos los nodos de un grafo
- Funcionamiento:
  - Se elige algún nodo para comenzar
  - Se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

Recorrido en anchura

```
procedimiento recorridoa(G)
    para cada v ϵ N hacer marca[v] ← no visitado
    para cada v ϵ N hacer
        si marca[v] ≠ visitado entonces ra(v)
procedimiento ra(v)
    Q ← cola vacía
    marca[v] ← visitado
    poner v en Q
    mientras Q no esté vacía hacer
        u ← primero(Q)
        quitar u de Q
        para cada nodo w adyacente a u hacer
            si marca[w] ≠ visitado entonces
                marca[w] ← visitado
                poner w en Q
```

Recorrido en anchura - Ejemplo



- Si el recorrido empieza por el nodo 1 y suponemos que los nodos adyacentes se examinan en orden numérico, el recorrido en anchura será:
  - 1, 2, 3, 4, 5, 6, 7, 8

Recorrido en anchura - Coste

- Suponemos un grafo de n nodos y m aristas.
- El tiempo de ejecución será $\Theta(\max(n,m))$

ÁRBOL DE RECUBRIMIENTO MÍNIMO

Árbol de recubrimiento mínimo

- Sea G=<N,A> un grafo conexo no dirigido.
- Las aristas tienen una longitud no negativa.
- Objetivo:
  - Hallar un subconjunto T de las aristas de G tal que utilizando solamente las aristas de T, todos los nodos deben quedar conectados, y además la suma de las longitudes de las aristas de T debe ser tan pequeña como sea posible.
- T debe tener n-1 aristas
- G'=<N,T> se denomina árbol de recubrimiento mínimo para el grafo G.
- Tiene muchas aplicaciones. Por ejemplo:
  - Red de comunicaciones más barata para conectar un conjunto de ciudades

Algoritmo de Prim

- Es un algoritmo voraz
- El algoritmo utiliza dos conjuntos:
  - B es un conjunto de nodos
  - T es el conjunto de aristas.
- Empezamos por un nodo arbitrario.
- En cada paso:
  - Se busca la arista más corta posible {u,v} tal que u ϵ B y v ϵ N-B
  - La arista se añade a T y el nodo v se añade a B

Algoritmo de Prim

función Prim(G=<N,A>): conjunto aristas

T ← Ø

B ← {un nodo arbitrario de N}

mientras B ≠ N **hacer**

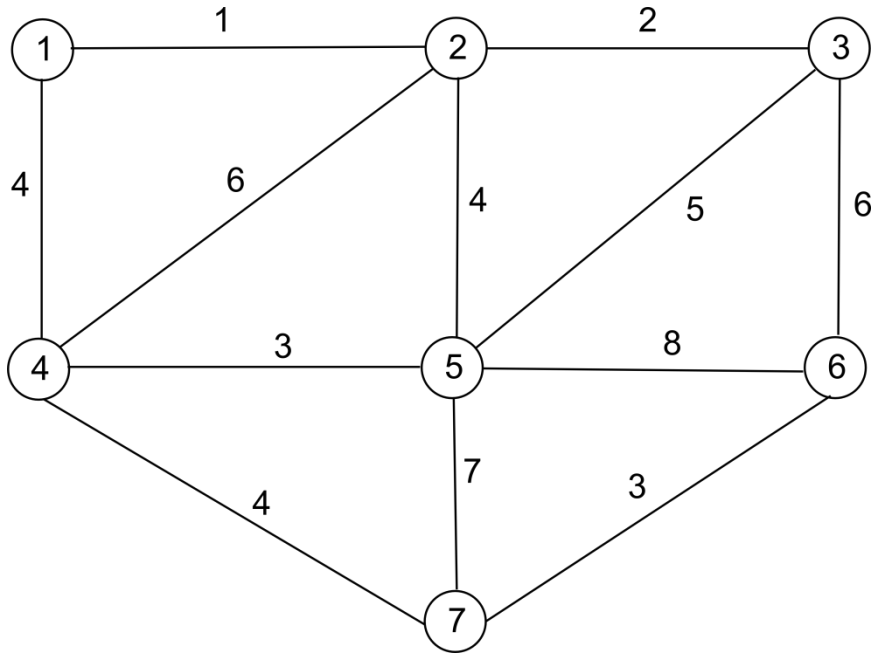
buscar e= {u,v} de longitud mínima tal que u ϵ B y v ϵ N-B

T ← T U {e}

B ← B U {v}

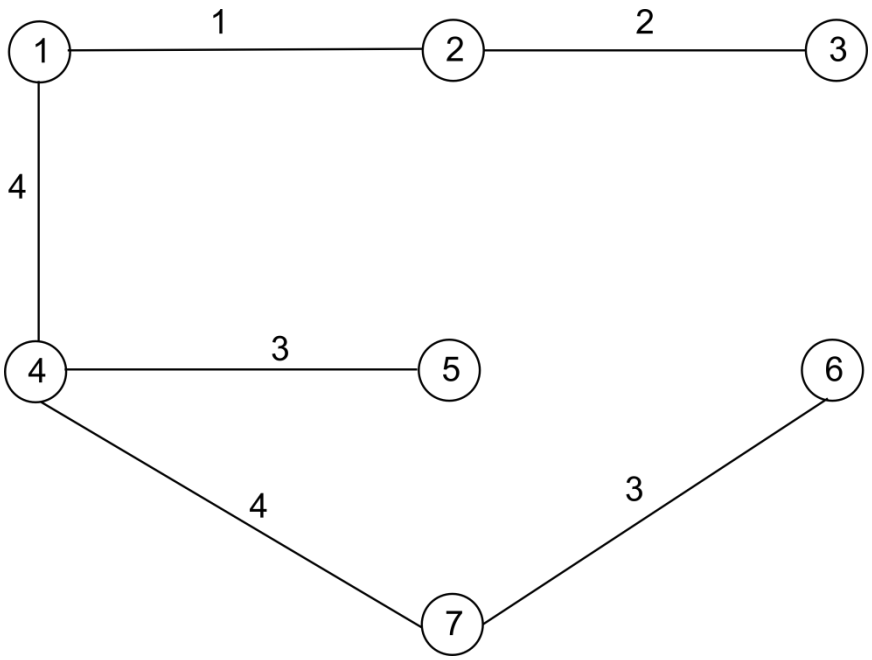
devolver T

Algoritmo de Prim - Ejemplo



Paso	{u,v}	B
	--	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

Algoritmo de Prim - Ejemplo



Algoritmo de Prim - Implementación

- Los nodos de G están numerados de 1 a n.
- La matriz L nos da la longitud de las aristas, con L[i,j] = ∞ si no existe arista.
- Usaremos dos vectores:
  - Para todo nodo i ϵ N-B:
    - más_próximo[i] proporciona el nodo de B más próximo a i
    - distmin[i] proporciona a distancia desde i hasta ese nodo.
  - Para todo nodo i ϵ B:
    - distmin[i] = -1
    - Esto nos permite saber los nodos que están en B.

Algoritmo de Prim - Implementación

```
función Prim (L[1..n,1..n]): conjunto de aristas
T ← Ø
para i ← 2 hasta n hacer
    más_próximo[i] ← 1
    distmin[i] ← L[ i,1]
repetir n-1 veces
    min ← ∞
    para j ← 2 hasta n hacer
        si 0 ≤ distmin[ j ] < min entonces  min ← distmin[ j ]
                                            k ← j
    T ← T U { más_próximo[k],k }
    distmin[k] ← -1          // se añade k a B
    para j ← 2 hasta n hacer
        si L[ j,k ] < distmin[ j ] entonces   distmin[ j ] ← L[ j,k ]
                                              más_próximo[ j ] ← k
devolver T
```

Algoritmo de Prim - Coste

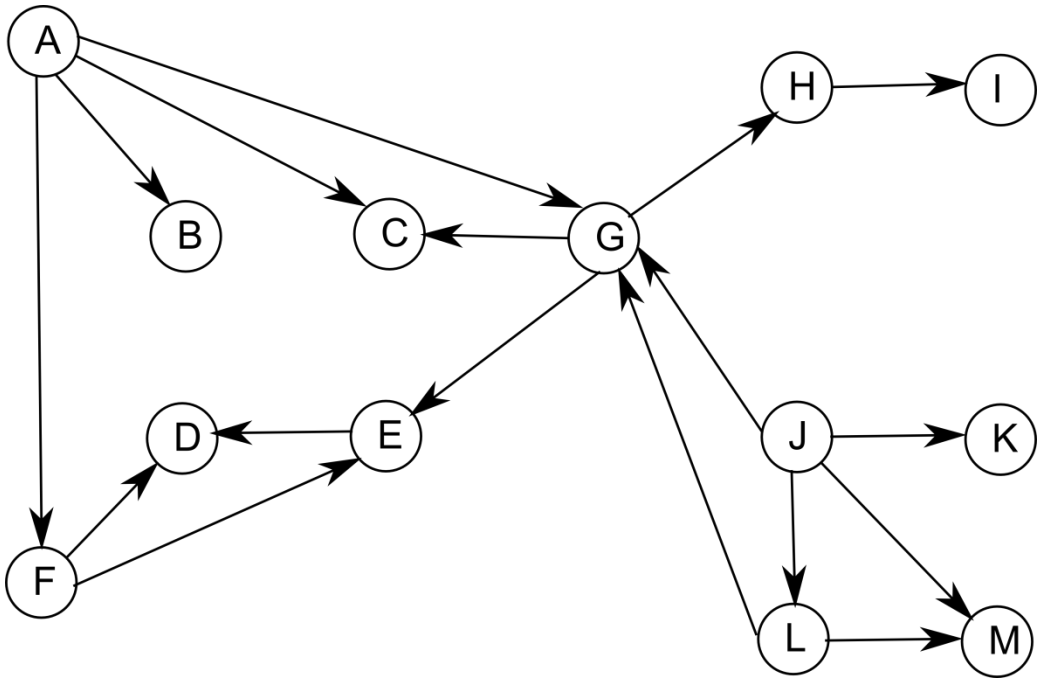
- El algoritmo de Prim requiere un tiempo que está en $\Theta(n^2)$

ORDENACIÓN TOPOLÓGICA

Ordenación topológica

- La ordenación topológica:
  - Se aplica a grafos acíclicos dirigidos o DAG (Directed Acyclic Graph).
  - Consiste en procesar todos los nodos del grafo de manera que ningún nodo se procese antes de cualquier nodo que apunta hacia él.
  - No es única:
    - Puede haber varias ordenaciones topológicas de un grafo.
- Una aplicación típica:
  - Si los nodos representan tareas
  - Hay una precedencia a la hora de ejecutar dichas tareas.
  - La ordenación topológica es una lista en orden lineal en que deben realizarse las tareas.

Ordenación topológica



- Una posible ordenación topológica:
  - J, K, L, M, A, G, H, I, F, E, D, B, C

Ordenación topológica

- **1.- Recorrido primero en profundidad ligeramente modificado**

```
procedimiento recorridop(G)
    para cada v ϵ N hacer marca[v] ← no visitado
    para cada v ϵ N hacer
            si marca[v] ≠ visitado entonces rp(v)
procedimiento rp(v)
    marca[v] ← visitado
    para cada nodo w adyacente a v hacer
            si marca[w] ≠ visitado entonces rp(w)
    escribir v
```

- **2.- Invertir el orden de la lista resultante.**