Tema 4. GRAFOS

ESTRUCTURAS DE DATOS Y ALGORITMOS II
GRADO EN INGENIERÍA INFORMÁTICA

María José Polo Martín

mjpolo@usal.es

Curso 2019-2020

VNiVERSiDAD D SALAMANCA
CAMPUS DE EXCELENCIA INTERNACIONAL

800 AÑOS 1218 ~ 2018

DEPARTAMENTO DE INFORMÁTICA Y AVTOMÁTICA

Tema 4. GRAFOS

1. Nivel abstracto o de definición
2. Nivel de Representación
3. Recorridos en grafos
4. Ordenación topológica
5. Algoritmos de Caminos de Coste Mínimo
   - Grafos no ponderados
   - Grafos Ponderados. Algoritmo Dijkstra
6. Árbol de expansión de coste mínimo
   - Algoritmo Prim
   - Algoritmo de Kruskal

1 NIVEL ABSTRACTO O DE DEFINICIÓN

- Intuitivamente, un **grafo** es un conjunto de puntos (vértices o nodos) y un conjunto de líneas (aristas o arcos), cada una de las cuales une un punto con otro
- Un grafo está completamente definido por su conjunto de vértices, V; y su conjunto de aristas, A

$$\mathbf{G = (V, A)}$$

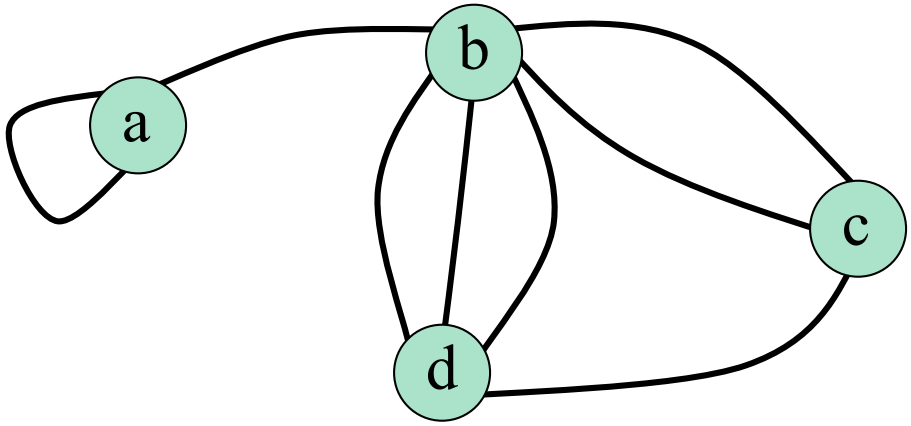
- **Orden** del grafo: número de vértices
- Cada **arista** es un par (v,w) donde v ,w $\in$ V
  - Si el par está ordenado $\Rightarrow$ **grafo dirigido** o **digrafo**
  - Un vértice w es **adyacente** a otro vértice v si y sólo si la arista (v,w) $\in$ A
  - En un grafo no dirigido con arista (v,w), y por tanto, (w,v), w es adyacente a v y v es adyacente a w
  - A veces una arista tiene un tercer componente, peso o costo $\Rightarrow$ grafos con aristas ponderadas

Grafos generales

- Puede haber varios arcos conectando dos vértices
- Algunos pares de vértices pueden estar desconectados
- Algunos arcos pueden conectar un vértice a sí mismo
  - Ejemplo: G = (V, A) donde

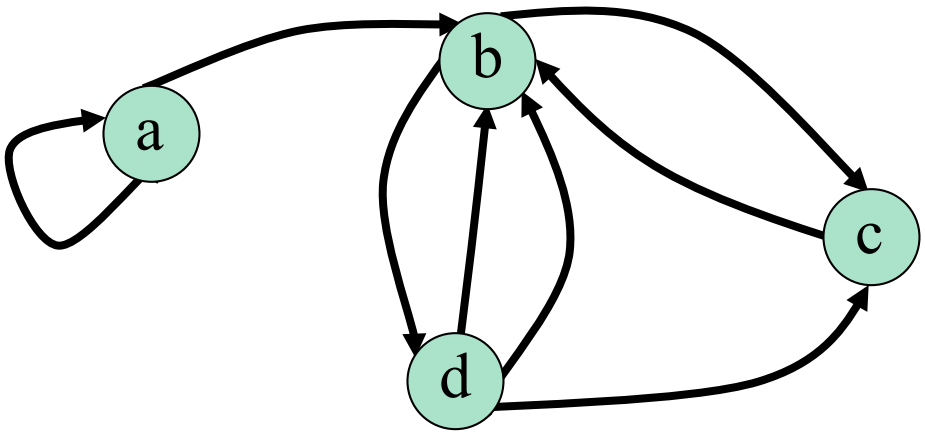
    V = {a, b, c, d }

    A = {(a,a), (a,b), (b,c), (b,c), (b,d), (b,d), (b,d), (c,d)}



Grafos DIRIGIDOS o DIGRAFOS

- Se impone un orden o dirección en las aristas del grafo
  - Ejemplo: G = (V, A) donde
    V = {a, b, c, d }
    A = {(a,a), (a,b), (b,c),(c,b), (b,d), (d,b), (d,b), (d,c)}



Caminos o trayectorias en grafos

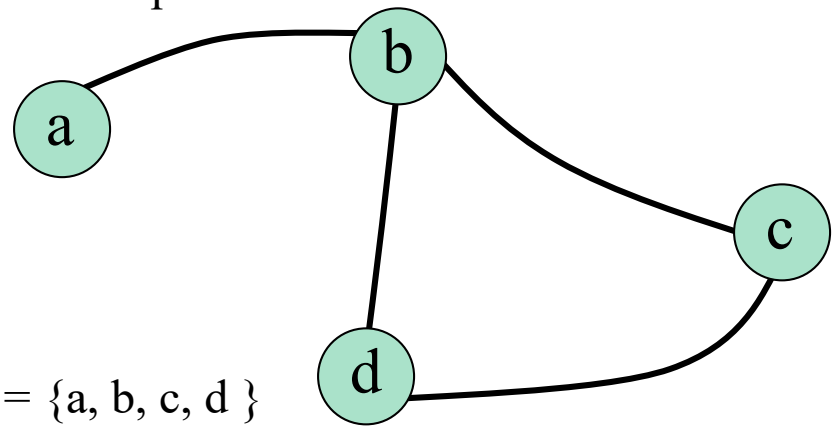
- Un **camino** o **trayectoria** en un grafo es una secuencia de vértices $v_{i(1)}$, $v_{i(2)}$, ..., $v_{i(n)}$ tal que la arista $(v_{i(x)},v_{i(x+1)}) \in A$ para $1 \leq x < n$
- **Longitud** de camino: número de arcos que lo componen (n-1)
- En un grafo se permiten caminos de un vértice a sí mismo:
  - Caso especial: un camino de un vértice a sí mismo que no contiene aristas tiene longitud cero
  - Si el grafo contiene una arista (v,v) de un vértice a sí mismo $\Rightarrow$ al camino v,v se le denomina bucle
- **Camino simple**: todos los vértices son distintos, excepto el primero y el último que pueden ser el mismo
- **Ciclo**: camino simple donde el vértice inicial y final coinciden

Grafos simples, acíclicos y conexos

- Un grafo G = (V, A) se denomina grafo **simple** si:
  1. No tiene bucles $\Rightarrow$ no existe en A un arco de la forma (v,v) siendo $v \in V$
  2. No hay más de una arista uniendo un par de vértices $\Rightarrow$ no existe más de una arista en A de la forma (v_i,v_j), para cualquier par de elemetos v_i, $v_j \in V$
- Un grafo G se denomina **acíclico** si no contiene ciclos
- Un grafo G se denomina **conexo** si hay un camino desde cualquier vértice a cualquier otro. No puede dividirse en dos sin eliminar uno de sus arcos
- Árbol: grafo conexo, simple y acíclico

Ejemplos

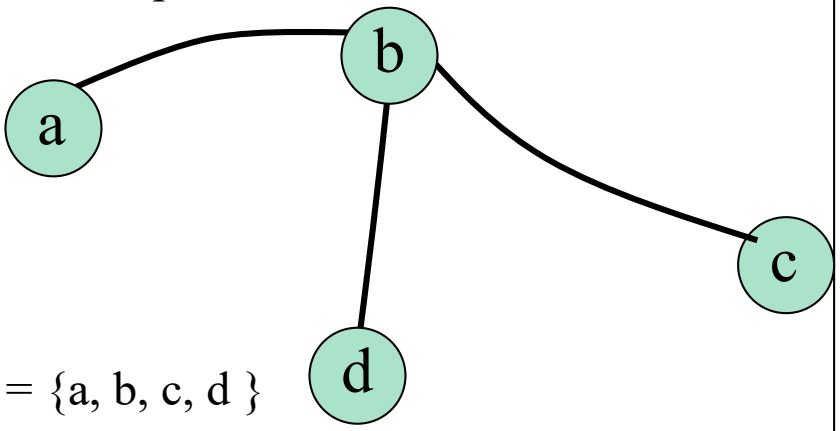
Grafo simple cíclico conexo



V = {a, b, c, d }

A = {(a,b), (b,c), (b,d), (d,c)}

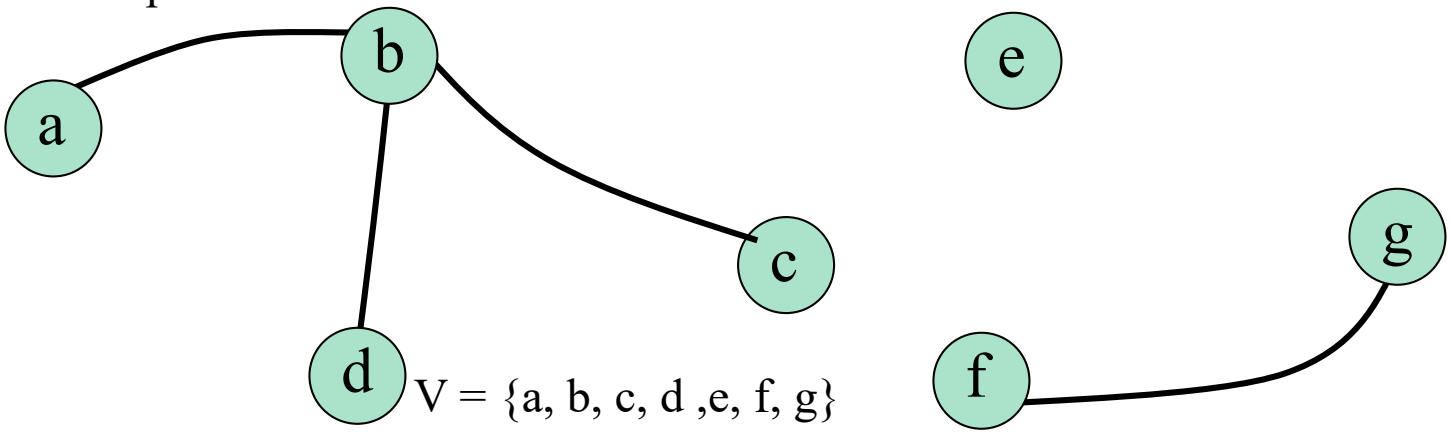
Grafo simple acíclico conexo



V = {a, b, c, d }

A = {(a,b), (b,c), (b,d)}

Grafo simple acíclico no conexo



V = {a, b, c, d ,e, f, g}

A = {(a,b), (b,c), (b,d), (f,g)}

Grado de un VÉRTICE

- Grafo no dirigido
  - **Grado de un vértice**: número de aristas que confluyen en él

- Grafo dirigido
  - **Grado interno o de entrada de un vértice:** número de aristas que terminan en él
  - **Grado externo o de salida de un vértice:** número de aristas que salen de él
  - **Grado de un vértice**: suma de sus grados interno y externo

Ejemplos de aplicación de grafos:

- Aeropuertos ⇒ Vértices: CIUDADES
  Aristas: - vuelos aéreos de una ciudad a otra
  - distancia entre ciudades (aristas ponderadas)
- Flujo de tráfico ⇒ Vértices: intersección de calles
  Aristas: conjunto de calles
  Peso aristas: límite de velocidad, número de carriles, etc.
- Programas ⇒ Vértices: bloques básicos de un programa
  Aristas: posibles transferencias de control de flujo
- Normalmente se hace referencia a los vértices de los grafos por sus etiquetas o identificadores
- En las aplicaciones reales, un vértice puede contener cualquier tipo de información, aunque en su estudio lo ignoremos
- Simplificamos el problema suponiendo que los identificadores de los vértices están numerados de 1 a n. Si no es así habrá que definir una función biyectiva que traduzca los identificadores al conjunto *{1, 2, ..., n}*

2 NIVEL DE REPRESENTACIÓN

- Dado un grafo $G=(V, A)$ de orden n, para $n \geq 1$; existen dos formas principales de representación:

  - MATRIZ DE ADYACENCIA
    - El grafo se representa mediante una matriz lógica ***m***, de dimensión nxn, donde ***m[i,j]*** es verdadero si y sólo si el arco (v_i, v_j) está en A

  - LISTAS DE ADYACENCIA
    - El grafo se representa mediante una matriz ***m*** de dimensión n, donde ***m[i]*** es un puntero a una lista enlazada que contiene todos los vértices adyacentes a v_i

Declaraciones básicas MATRIZ de ADYACENCIA

- El grafo se representa mediante una matriz lógica ***m***, de dimensión *nxn*, donde ***m[i,j]*** es verdadero si y sólo si el arco (v_i, v_j) está en A

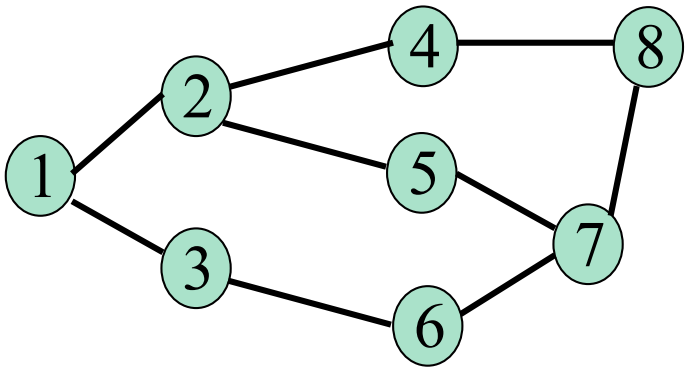
1. Grafos no poderados

```
tipo
   tipoGrafo: matriz[1..n, 1..n] de tipoLógico
fin tipo
```

2. Grafos ponderados

```
tipo
   tipoGrafo: matriz[1..n, 1..n] de tipoPeso
fin tipo
```

Ejemplo grafo no dirigido

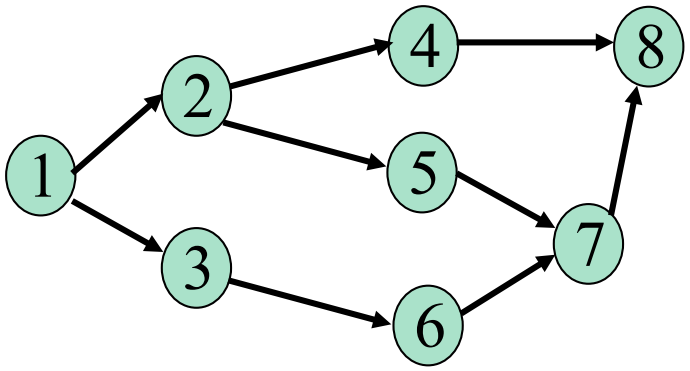


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	1	0	0	0
3	1	0	0	0	0	1	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	1	0
6	0	0	1	0	0	0	1	0
7	0	0	0	0	1	1	0	1
8	0	0	0	1	0	0	1	0

$$grado(v_i) = \sum_{j=1}^{n} m[i,j] = \sum_{j=1}^{n} m[j,i] \Rightarrow \sum filas \text{ ó } \sum columnas$$

$$grado(7) = \sum_{j=1}^{8} m[7,j] = 3$$

Ejemplo grafo dirigido



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	0	0	0	1	1	0	0	0
3	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

$$gradoInterno(v_i) = \sum_{k=1}^{n} m[k,i] \Rightarrow \sum filas$$

$$gradoExterno(v_i) = \sum_{k=1}^{n} m[i,k] \Rightarrow \sum columnas$$

$$grado(v_i) = gradoInterno(v_i) + gradoExterno(v_i)$$

$$grado(7) = \sum_{K=1}^{8} m[k,7] + \sum_{k=1}^{8} m[7,k] = 2+1 = 3$$

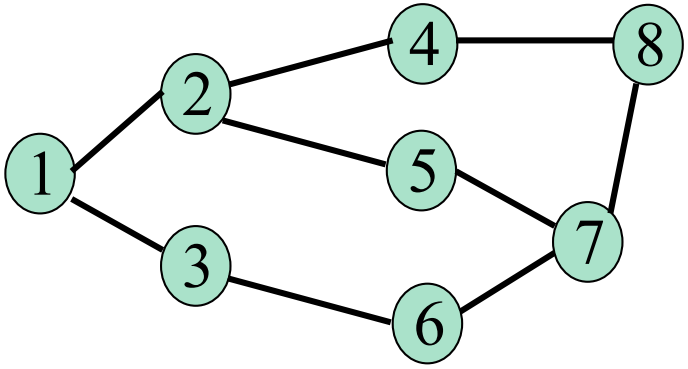
Representación mediante LISTAS DE ADYACENCIA

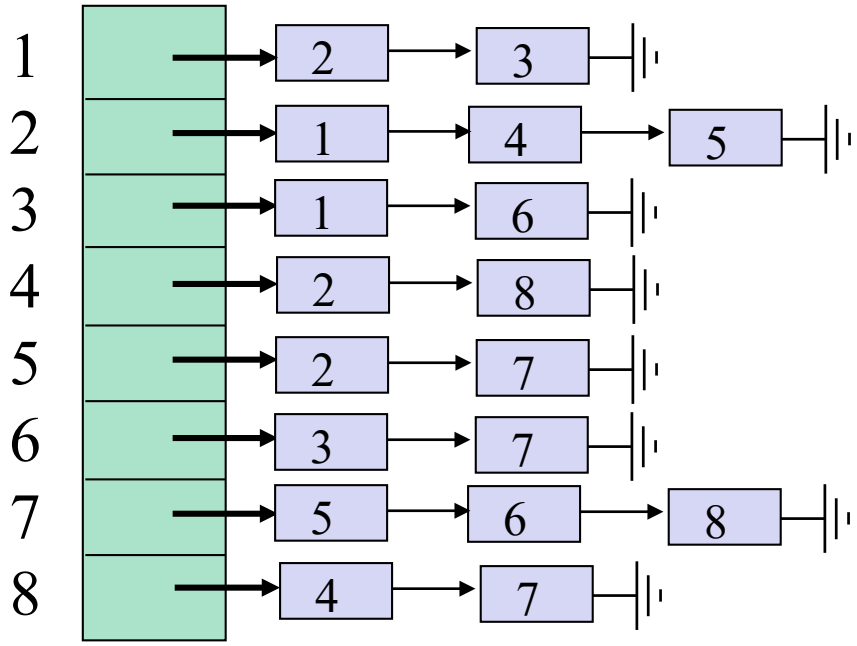
- Esta representación requiere dos estructuras de datos, una para representar los vértices y otra para representar los arcos:
  - **ARCOS**
    - **Listas de adyacencia:** a cada vértice se le asocia una lista que contiene todos sus vértices adyacentes
    - Si los arcos están ponderados habrá que reservar espacio en la estructura que representa los arcos para los **pesos**
  - **VÉRTICES**
    - **Directorio de vértices**: matriz que contiene una entrada para cada vértice del grafo, donde la entrada del vértice *i* apunta a una lista enlazada que contiene todos los vértices adyacentes a *i*
  - **Orden** del grafo: número de vértices

Declaraciones básicas

```
constante  MAXIMO = 100
tipos
        tipoArco = registro
                    vértice: tipoIdVértice
                    peso:    tipoPeso              // aristas ponderadas
                    sig : ↑tipoAraco
          fin registro
          punteroArco = ↑tipoAraco
          tipoGrafo = registro
             directorio : matriz[1..MÁXIMO] de punteroArco
             orden: entero
          fin registro
fin tipos
```

Ejemplo grafo no dirigido

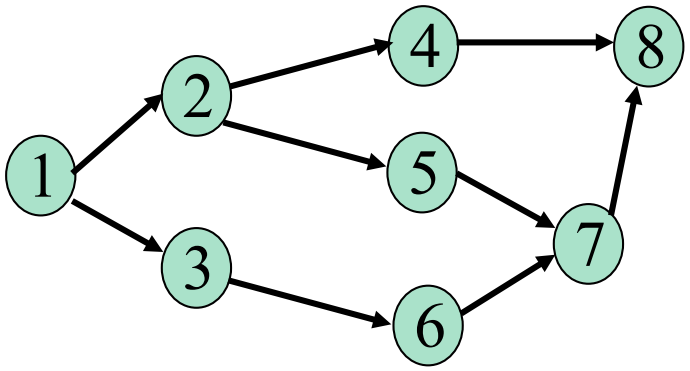


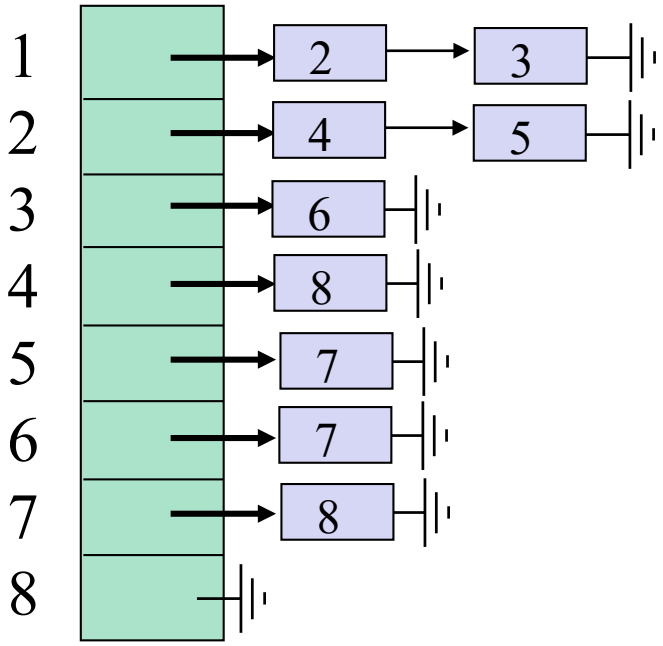


grado(v) = número de elementos en la lista de adyacencia de v

grado(7) = 3

Ejemplo grafo dirigido





gradoEntrada(v):	número de veces que v aparece en las listas de adyacencia del resto de los vértices
gradoSalida(v):	número de elementos en la lista de adyacencia de v

grado(7) = 2 +1 = 3

Observaciones

- La selección apropiada de la representación dependerá de las operaciones que se apliquen a los vértices y a los arcos del grafo
- Matriz de adyacencia
  - el tiempo de acceso requerido a un elemento es independiente del tamaño de V y A. Puede ser útil en aquellos algoritmos que necesiten saber si un arco determinado está presente en el grafo
  - La principal desventaja es que requiere un espacio proporcional a n^2 para representar todos los arcos posibles, aun cuando el grafo tenga menos de n^2 arcos (ocurre siempre en el caso de grados dirigidos)
  - Examinar la matriz llevará un tiempo de $O(n^2)$
- Listas de adyacencia
  - Requieren un espacio proporcional a la suma del número vértices y de arcos, es útil cuando se quieren representar grafos que tienen un número de arcos mucho menor que n^2
  - Una desventaja potencial es, que determinar si existe un arco, puede llevar un tiempo del $O(n)$, ya que puede haber n vértices en lista de adyacencia
- **En el resto del tema utilizaremos la representación mediante listas de adyacencia**

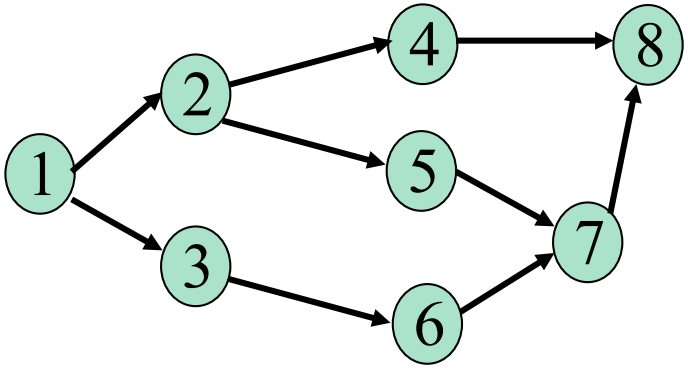
3 RECORRIDOS EN GRAFOS

- Para resolver muchos problemas relacionados con grafos, es necesario visitar los vértices y los arcos de manera sistemática
- Debe elegirse un vértice de partida y desde él visitar el resto de forma que cada vértice se visite una sola vez
- Un vértice puede encontrarse varias veces en el recorrido, es necesario por tanto, marcar cada vértice a medida que se visita, para no volver a visitarlo $\Rightarrow$ modificación en la declaración de las estructuras básicas

- Dos categorías básicas:
  - Recorrido en **Amplitud**
  - Recorrido en **Profundidad**

Recorridos y ejemplos

- Recorrido en **Amplitud**
  - Se visita un vértice inicial, a continuación todos sus vértices adyacentes, después los adyacentes a estos últimos y así sucesivamente hasta que todos hayan sido visitados
- Recorrido en **Profundidad**
  - Se visita un vértice inicial y se siguen visitando sus vértices en una trayectoria hasta el final de esa trayectoria, después se elige otra trayectoria y se visitan todos sus vértices hasta el final de la trayectoria y así sucesivamente hasta visitar todos los vértices

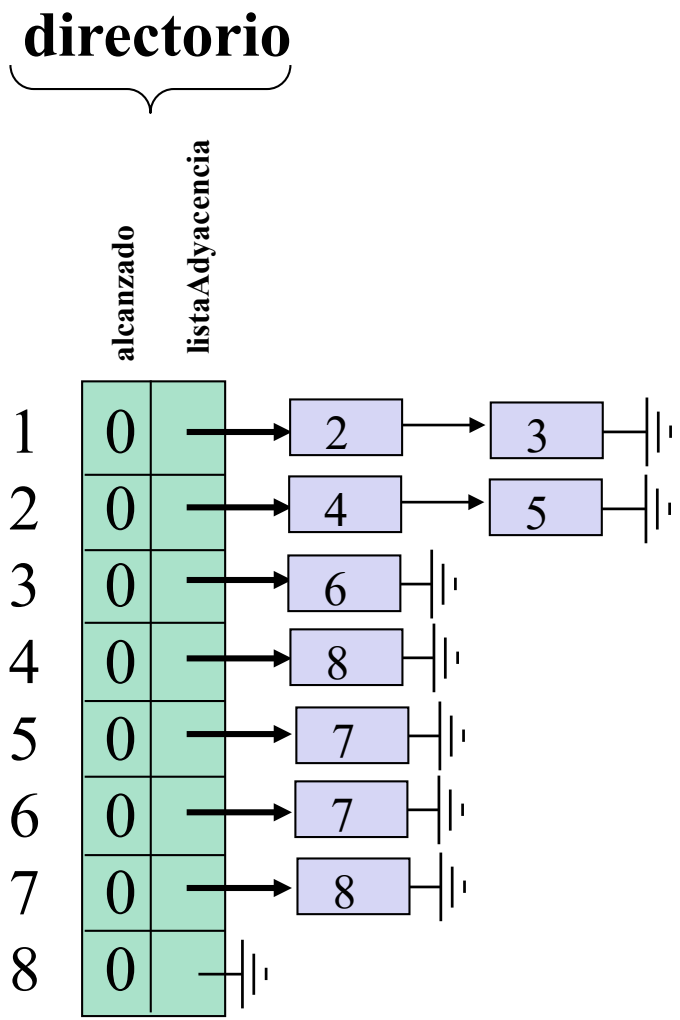


Recorrido en AMPLITUD ⇒1 2 3 4 5 6 8 7

Recorrido en PROFUNDIDAD⇒1 2 4 8 5 7 3 6

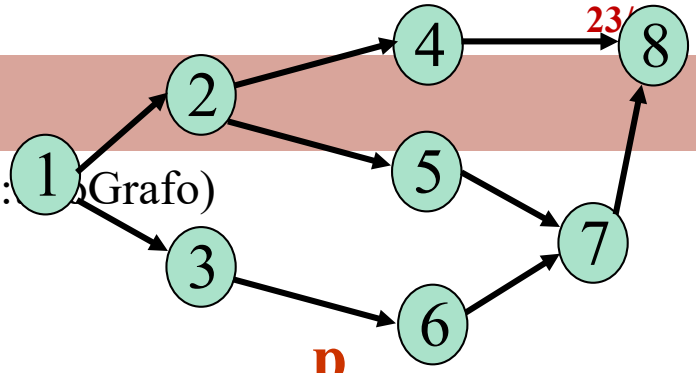
Modificación declaraciones básicas LISTAS DE ADYACENCIA

```
constante  MÁXIMO = 100
tipos
tipoArco = registro
    vértice: tipoIdVértice
    peso: tipoPeso     // aristas ponderadas
    sig : ↑tipoArco
fin registro
   punteroArco = ↑ARCO
tipoVértice = registro
    alcanzado: tipoLógico
    ...
    listaAdyacencia: punteroArco
fin registro
tipoGrafo = registro
    directorio : matriz[1..MÁXIMO] de tipoVértice
    orden: entero
 fin registro
```

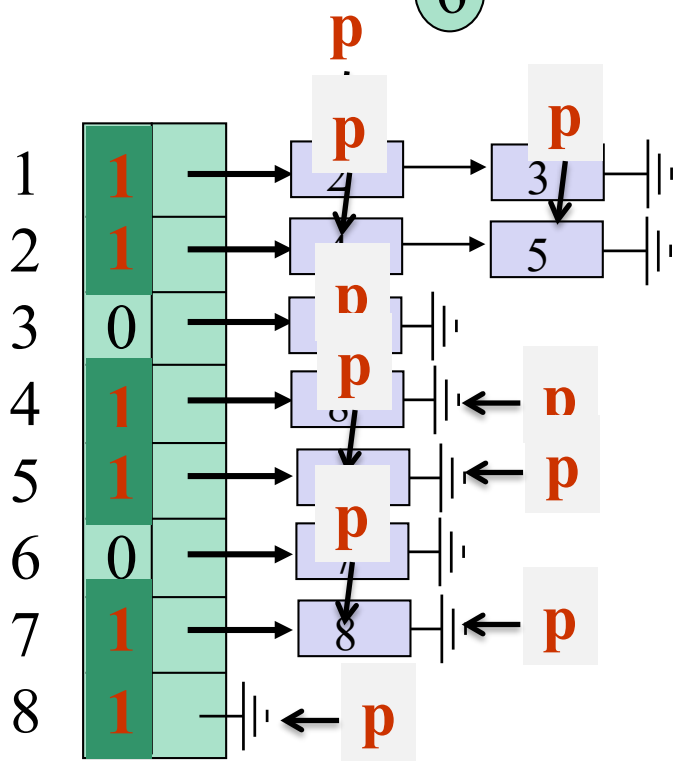


DEPARTAMENTO DE INFORMÁTICA Y AUTOMÁTICA

Algoritmo de recorrido en PROFUNDIDAD



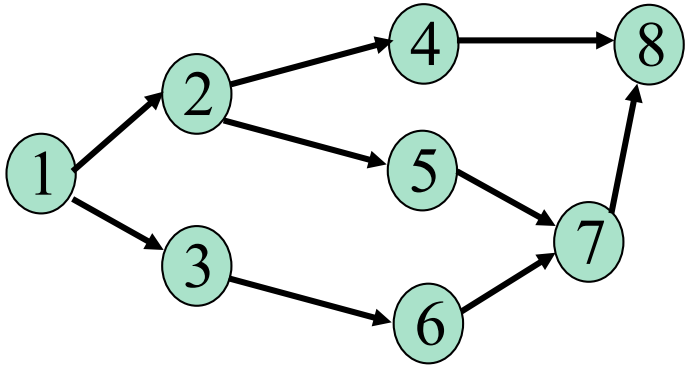

```
procedimiento profundidad(vInicio: tipoIdVértice, ref g: tipoGrafo)
1.  w: tipoIdVértice
2.  p : punteroArco
3.  visitar(vInicio)
4.  g.directorio[vInicio].alcanzado ←VERDADERO
5.  p ← g.directorio[vInicio].listaAdyacencia
6.     mientras p ≠ NULO hacer
7.           w ← p↑.vértice
8.           si NOT(g.directorio[w].alcanzado) entonces
9.              profundidad(w, g)
10.          fin si
11.          p ← p↑.sig
12.    fin mientras
```

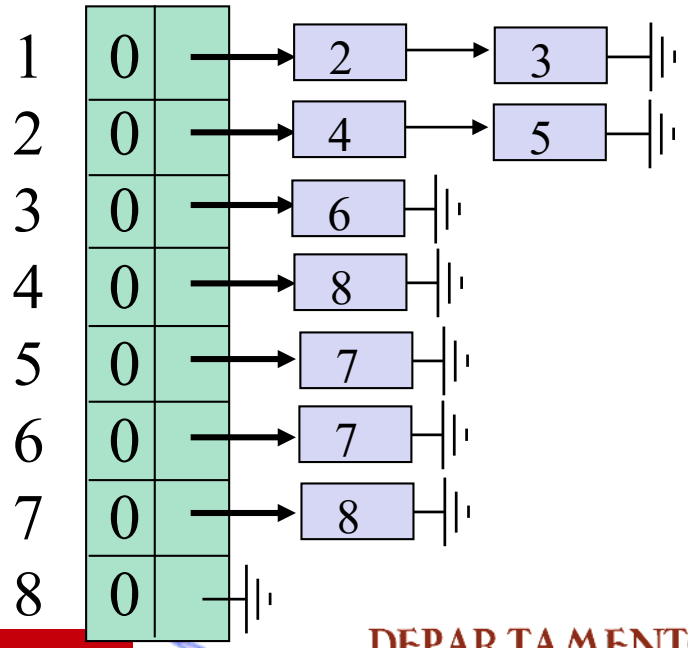


profundidad(1,g) ⇒ 1 2 4 8 5 7 …

Algoritmo de recorrido en AMPLITUD

```
procedimiento amplitud(vInicio: tipoIdVértice, ref  g: tipoGrafo)
1.   w: tipoIdVértice
2.   p: punteroArco
3.   c: tipoCola
4.    creaVacía(c)
5.    inserta(vInicio,c)
6.     mientras NOT(vacia(c)) hacer
7.        w ← suprime(c)
8.        si NOT (g.directorio[w].alcanzado) entonces
9.            visitar(w)
10.           g.directorio[w].alcanzado ←VERDADERO
11.           p ← g.directorio[w].listaAdyacencia
12.           mientras p≠ NULO hacer
13.              w ← p↑.vértice
14.              inserta(w,c)
15.              p ← p↑.sig
16.           fin mientras
17.       fin si
18.    fin mientras
```





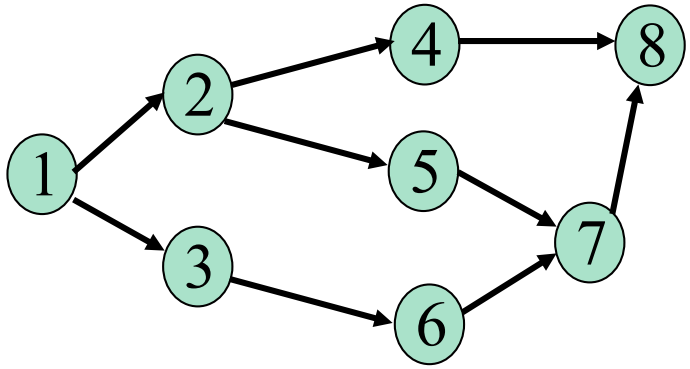
Observaciones

- El directorio de vértices tiene que estar correctamente inicializado antes de comenzar el recorrido: **ningún vértice marcado**
- Puede que algunos vértices no se visiten dependiendo del vértice de partida y del tipo de grafo:
  - En grafos dirigidos si existen vértices inalcanzables desde el vértice inicial
  - En grafos no conexos el recorrido solo visita los vértices conectados con el vértice inicial

- **Solución**: buscar los vértices no marcados y aplicarles de nuevo el recorrido hasta que no queden vértices sin marcar

4 ORDENACIÓN TOPOLÓGICA

- Consiste en la clasificación de los vértices de un grafo dirigido acíclio (gda) tal que si existe un camino de v a w , v aparece antes que w en la clasificación
- Observaciones:
  - No es posible la ordenación topológica si el grafo tiene ciclos (para dos vértices v y w en el ciclo, v precede a w y w precede a v)
  - La ordenación topológica no es necesariamente única



Orden TOPOLÓGICO:
1) 1 2 3 4 5 6 7 8
2) 1 3 2 6 4 5 7 8

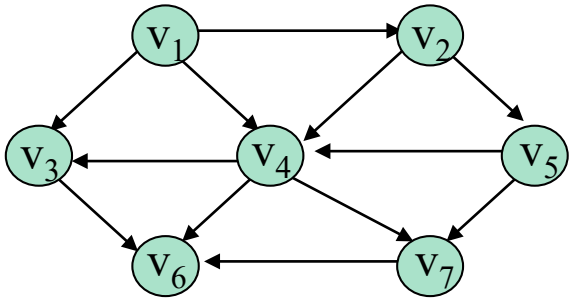
Algoritmo sencillo para establecer la ordenación topológica:

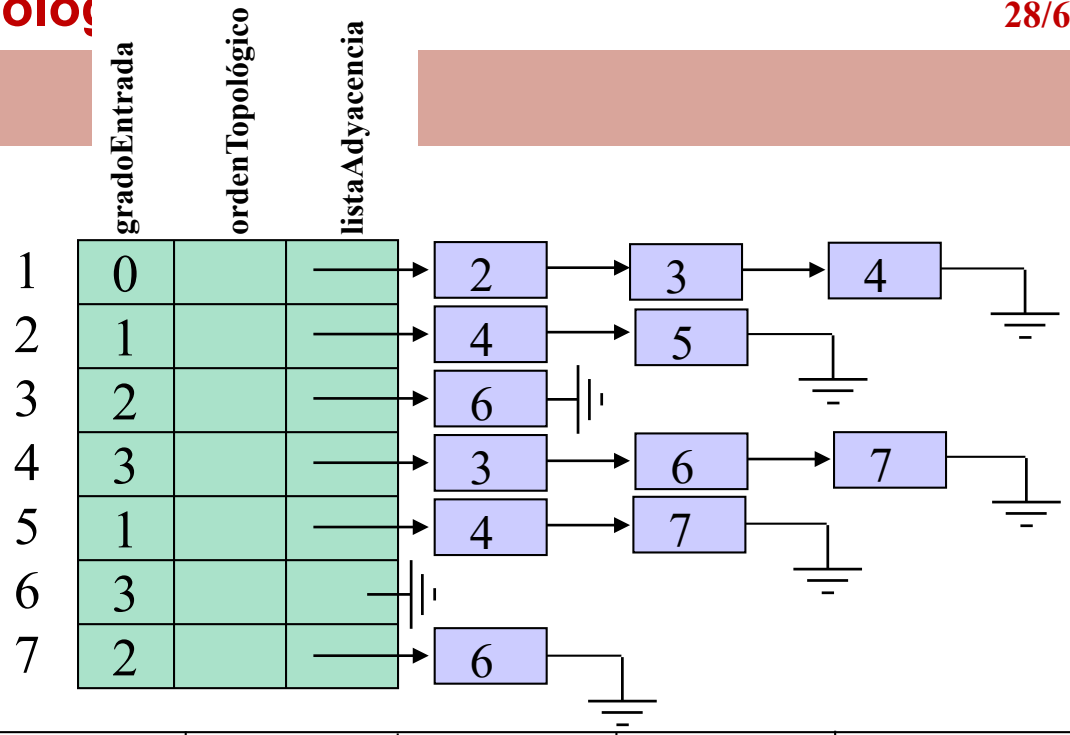
1. Encontrar un vértice cualquiera, *v*, con grado de entrada cero (a ese vértice no llegan aristas) y asignarle el primer orden topológico
2. Decrementar el grado de entrada de todos los vértices adyacentes a *v* y aplicar la misma estrategia a aquellos vértices que todavía no tengan asignado un orden

   El primer vértice cuyo grado de entrada se convierta en cero, será el siguiente vértice en orden topológico, ya que no podremos volver a acceder a él desde *v*.

- Para formalizar el algoritmo es necesario calcular el grado de entrada de cada vértice del grafo y guardar esta información en la estructura que representa al grafo (ampliación declaraciones básicas)

Ejemplo:





Vértice ordenado	v_1	v_2	v_5	v_4	v_3	v_7	v_6
Orden topológico	1	2	3	4	5	6	7

1	0	1	1	1	1	1	1
2	1	0	2	2	2	2	2
3	2	1	1	0	5	5	5
4	3	2	0	4	4	4	4
5	1	1	0	3	3	3	3
6	3	3	3	2	1	0	7
7	2	2	1	0	0	6	6

Algoritmo de ORDENACIÓN TOPOLÓGICA (versión 1)

```
procedimiento ordenTopológico(ref g: tipoGrafo)
1.  orden: entero
2.  p : punteroArco
3.  v,w : tipoIdVértice
4.  iniciar(g)
5.   para orden = 1 hasta g.orden hacer
6.      v ← buscarVerticeGradoCeroNoOrdenTop(g)
7.      si v  =  −1 entonces error (“el grafo tiene un ciclo”)
8.      sino
9.         g.directorio[v].ordenTopológico ← orden
10.        p ← g.directorio[v].listaAdyacencia
11.        mientras p ≠ NULO hacer
12.           w ← p↑.vértice
13.           g.directorio[w].gradoEntrada ← g.directorio[w].gradoEntrada − 1
14.           p ← p↑.sig
15.        fin mientras
16.     fin si
17.  fin para
```

Observaciones:

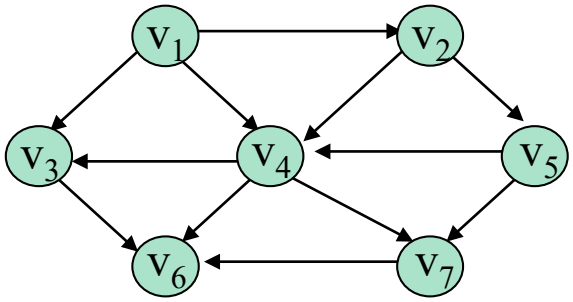
- La función ***buscarVerticeGradoCeroNoOrdenTop*** recorre el directorio de vértices buscando un vértice ***v*** con grado de entrada cero al que todavía no se haya asignado un orden topológico. Devuelve el identificador de ese vértice, si existe; y –1 si no existe, indicando que hay un ciclo
- Cada llamada a esta función recorre secuencialmente el directorio de vértices y se le llama *n* veces ⇒ tiempo de ejecución deficiente de $O(n^2)$
- Mejora: utilización de una cola en la que se vayan insertando todos los vértices con grado cero.
    - Cada vez que se decrementan los grados de entrada de los vértices, estos se revisan y se insertan en la cola si su grado se convierte en cero.

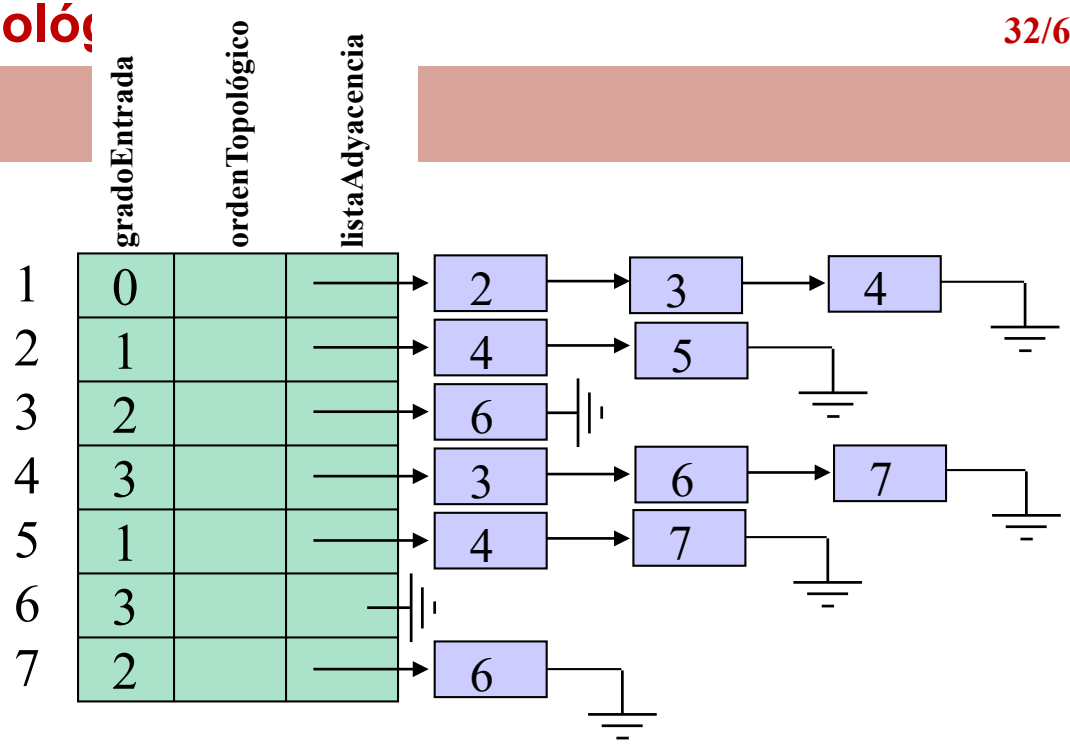
Proceso de ordenación topológica II

Para cada vértice se calcula su grado de entrada (paso previo)

1. Todos los vértices con grado de entrada cero se insertan en la cola (inicialmente vacía)
2. Mientras la cola no esté vacía
   - 2.1 Se suprime un vértice de la cola: ***v***.
   - 2.2 Se decrementan los grados de todos los vértices adyacentes a ***v***.
   - 2.3 Si el grado de entrada de algún vértice se convierte en cero, se inserta en la cola
3. La ordenación topológica coincide con el orden en que los vértices van saliendo de la cola

Ejemplo:





Vértice	grado de entrada antes de desencolar						
v_1	**0**	0	0	0	0	0	0
v_2	1 → **0**	0	0	0	0	0	0
v_3	2 → **1**	1	1 → **0**	0	0	0	0
v_4	3 → **2** → **1** → **0**	0	0	0	0	0	0
v_5	1	1 → **0**	0	0	0	0	0
v_6	3	3	3	3 → **2** → **1** → **0**			
v_7	2	2	2 → **1** → **0**	0	0	0	0
encolar	v_1	v_2	v_5	v_4	v_3, v_7		v_6
desencolar	v_1	v_2	v_5	v_4	v_3	v_7	v_6
Orden top.	1	2	3	4	5	6	7

Algoritmo de ordenación topológica (versión 2)

```
procedimiento ordenTopológico(ref g: tipoGrafo)
1.   c: tipoCola
2.   iniciar(g)
3.   creaVacía(c)
4.   para v = 1 hasta g.orden hacer
5.        si g.directorio[v].gradoEntrada = 0 entonces  inserta(v,c)  fin si
6.   fin para
7.   orden ← 1
8.   mientras NOT (vacia(c)) hacer
9.      v ← suprime(c)
10.     g.directorio[v].ordenTopológico ← orden
11.     orden  ← orden + 1
12.     p ← g.directorio[v].listaAdyacencia
13.     mientras p ≠ NULO hacer
14.          w ← p↑.vértice
15.          g.directorio[w].gradoEntrada←g.directorio[w].gradoEntrada – 1
16.          si g.directorio[w].gradoEntrada = 0 entonces  inserta(w,c) fin si
17.          p ←p↑.sig
18.     fin mientras
19. fin   mientras
```

Observaciones

- El tiempo de ejecución del nuevo algoritmo es $O(n+a)$
  - Las operaciones encola y desencola una vez por vértice, $O(n)$
  - El bucle mientras interior se ejecuta una vez por arista, $O(a)$
  - La asignación de valores iniciales toma un tiempo proporcional al tamaño del grafo

- Los algoritmos de ordenación topológica, con pequeñas modificaciones, pueden utilizarse para determinar si un grafo es cíclico

5 ALGORITMOS DE CAMINOS DE COSTE MÍNIMO

Planteamiento del Problema:

1 Grafo dirigido no ponderado:

- Determinar el camino de menor longitud (menor número de arcos) entre un par de vértices
- Determinar la trayectoria real que permite ir de un vértice a otro mediante el camino de menor longitud

2 Grafo dirigido ponderado: cada arco del grafo (v_i,v_j) tiene asociado un peso $p_{i,j} \geq 0$, de tal forma que el coste de un camino $v_1, v_2, ..., v_k$ es :

$$\text{coste}(v_1, v_k) = \sum_{i=1}^{k-1} p_{i,i+1}$$

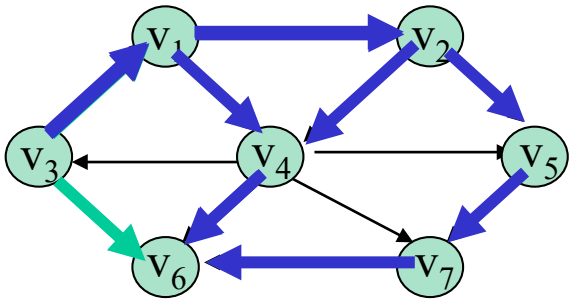
- Determinar el camino de coste mínimo entre un par de vértices
- Determinar la trayectoria real que nos conduce con coste mínimo de un vértice a otro

- Algoritmos proporcionan caminos mínimos y trayectorias entre un vértice inicial y el resto de los vértices del grafo

GRAFOS NO PONDERADOS

- Tomando un vértice inicial, *v*, encontrar el camino más corto de *v* al resto de los vértices del grafo
- Grafo no ponderado: solo interesa el número de aristas que contiene el camino. Es un caso especial de grafo ponderado con coste igual a 1 en todos los arcos

Ejemplo



Trayectoria del camino	Longitud, coste o distancia del camino
Un único camino simple de v_3 a v_1	
(v_3, v_1)	1
Varios caminos simples de v_3 a v_6	
(v_3, v_6)	1
(v_3, v_1, v_4, v_6)	3
$(v_3, v_1, v_2, v_4, v_6)$	4
$(v_3, v_1, v_2, v_5, v_7, v_6)$	5

Ampliación declaraciones básicas

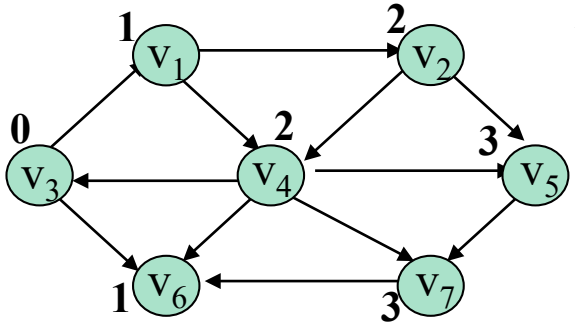
- Los algoritmos de caminos mínimos utilizan estrategias que necesitan ampliar la información para cada vértice:

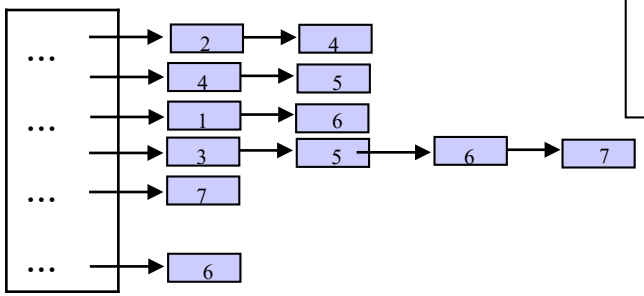
  - **alcanzado**: tendrá valor VERDADERO si en el recorrido se ha pasado por ese vértice. Inicialmente con valor FALSO

  - **distancia**: valor que indicará el número de arcos desde el vértice inicial al vértice representado (longitud de camino). Inicialmente con valor inalcanzable (*"infinito"*) excepto el vértice de partida que tendrá valor *0*

  - **anterior**: almacenará el último vértice desde el que se alcanza el vértice representado. Inicialmente a valor *0*. Permitirá mostrar los caminos reales

Ejemplo: vértice origen v_3





Estado inicial directorio de vértices

v	alcanzado	distancia	anterior
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

distanciaActual = 0 y noAlcanzado(v_3)

v	alcanzado	distancia	anterior
v_1	0	1	v_3
v_2	0	∞	0
v_3	1	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0

distanciaActual = 1 y noAlcanzado(v_1 y v_6)

v	alcanzado	distancia	anterior
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	1	1	v_3
v_7	0	∞	0

distanciaActual = 2 y noAlcanzado(v_2 y v_4)

v	alcanzado	distancia	anterior
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4

distanciaActual = 3 y noAlcanzado(v_5y v_7)

v	alcanzado	distancia	anterior
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4

Algoritmo de Camino de Longitud Mínima (versión 1)

```
procedimiento caminoMínimo1(vInicio:tipoIdeVertice,ref g: tipoGrafo)
1.    distanciaActual: entero
2.    v,w : tipoIdVértice
3.    p: punteroArco
4.    iniciar(g)
5.    g.directorio[vInicio].distancia ← 0
6.    g.directorio[vInicio].anterior ← 0
7.    para distanciaActual = 0 hasta g.orden – 1 hacer
8.        para v = 1 hasta g.orden hacer
9.          si (NOT(g.directorio[v].alcanzado) and
10.          (g.directorio[v].distancia=distanciaActual) entonces
11.              g.directorio[v] .alcanzado ← VERDADERO
12.              p ← g.directorio[v].listaAdyacencia
13.              mientras p ≠ NULO hacer
14.                  w ← p↑.vértice
15.                  si g.directorio[w].distancia = INFINITO entonces
16.                      g.directorio[w].distancia ← g.directorio[v].distancia + 1
17.                      g.directorio[w].anterior ← v
18.                  fin si
19.                  p ← p↑.sig
20.              fin mientras
21.         fin si
22.       fin para
23.   fin para
```

Observaciones:

- Debe inicializarse correctamente el directorio de vértices: todos los vértices son inalcanzables excepto el vértice de partida, cuya longitud de camino es 0
- Cuando se procesa un vértice se tiene la garantía de que no se encontrará un camino más económico para alcanzarlo. Alcanzado toma el valor 1 y el procesamiento de ese vértice está completo
- Pueden existir vértices inalcanzables desde el vértice inicial
  - ¿Qué ocurre si el vértice de inicio es v_6?
- Es posible mostrar el camino real de un vértice a otro regresando a través de la variable anterior

v	alcanzado	distancia	anterior
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4

¿existe un camino entre v_3 y v_7?

Si, existe un camino, que además es el de coste mínimo (distancia 3)

¿cuál es la trayectoria de ese camino mínimo?

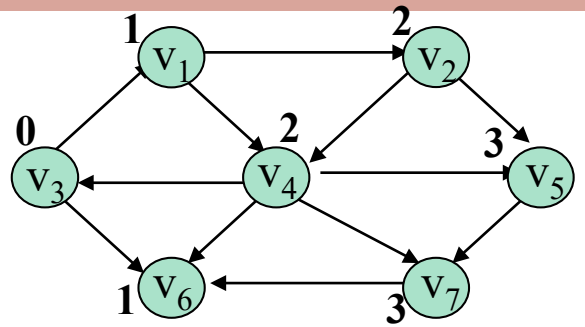
v_3 v_1 v_4 v_7

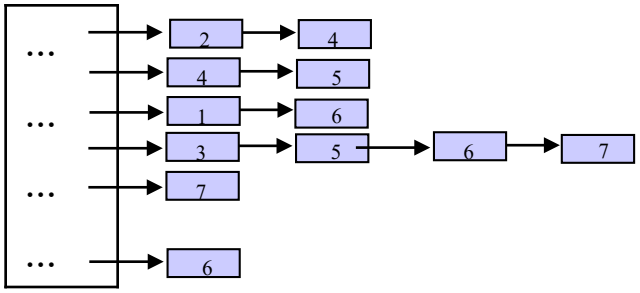
Análisis

- El tiempo de ejecución del algoritmo es bastante deficiente debido a los ciclos "para" doblemente anidados($O(n^2)$)
- El ciclo externo continúa hasta *n-1* aunque todos los vértices se hayan alcanzado mucho antes

- MEJORA
  - Utilizar una cola que inicialmente guardará el vértice de partida (longitud de camino 0), y a medida que va sacando los vértices de la cola almacena sus adyacentes (longitud de camino 1), y así sucesivamente
  - La cola garantiza que no se procesan vértices de longitud de camino ***d+1*** hasta que no se hayan procesado todos los vértices con longitud de camino ***d***

Ejemplo: vértice origen v_3





Estado inicial dir.

v	alc.	dist.	ant.
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0
C	v_3		

v_3 desencolado

v	alc.	dist.	ant.
v_1	0	1	v_3
v_2	0	∞	0
v_3	1	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0
C	v_1 v_6		

v_1 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0
C	v_6 v_2 v_4		

v_6 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	1	1	v_3
v_7	0	∞	0
C	v_2 v_4		

v_2 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	∞	0
C	v_4 v_5		

v_4 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
C	v_5 v_7		

v_5 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
C	v_7		

v_7 desencolado

v	alc.	dist.	ant.
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4
C	vacía		

Algoritmo de Camino de Longitud Mínima (versión 2)

```
procedimiento caminoMínimo2(vInicio: tipoIdeVertice, ref g:tipoGrafo)
1.    p: punteroArco
2.    v, w: tipoIdVértice
3.    c: tipoCola
4.    iniciar(g)
5.    g.directorio[vInicio].distancia ← 0
6.    g.directorio[vInicio].anterior ←  0
7.    creaVacía(c )
8.    inserta(vInicio,c)
9.     mientras NOT (vacia(c)) hacer
10.       v ← suprime(c)
11.       g.directorio[v].alcanzado ←VERDADERO // No hace falta
12.       p ← g.directorio[v].listaAdyacencia
13.       mientras p ≠ NULO hacer
14.             w ← p↑.vértice
15.             si g.directorio[w].distancia = INFINITO entonces
16.                 g.directorio[w].distancia←g.directorio[v].distancia+1
17.                 g.directorio[w].anterior ← v
18.                 inserta(w,c)
19.             fin si
20.             p ← p↑.sig
21.       fin mientras
22.    fin mientras
```

Observaciones:

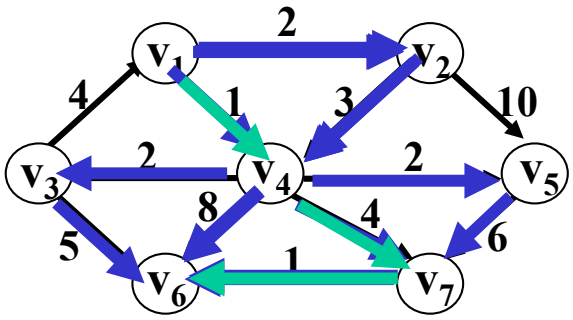
- Una vez procesado un vértice nunca puede volver a entrar en la cola
  - queda marcado implícitamente que no se debe volver a procesar
  - puede eliminarse la información alcanzado
- Puede ocurrir que la cola se vacíe prematuramente si algunos vértices son inalcanzables desde el vértice origen
  - para estos vértices inalcanzables se obtendrá una distancia "infinita", lo cual es perfectamente lógico
  - ¿Qué ocurre si el vértice de inicio es v_6?
- El tiempo de ejecución de este algoritmo es $O(n+a)$

GRAFOS PONDERADOS

- Tomando un vértice inicial, *v*, encontrar el camino de menor coste (puede que no sea el más corto) de *v* al resto de los vértices del grafo

Ejemplo

Trayectoria del camino	Longitud, coste o distancia del camino
Varios caminos simples de v_1 a v_6	
$(v_1, v_2, v_4, v_3, v_6)$	12
(v_1, v_2, v_4, v_6)	13
$(v_1, v_2, v_4, v_5, v_7, v_6)$	14
$(v_1, v_2, v_4, v_7, v_6)$	10
(v_1, v_4, v_6)	9
…	
(v_1, v_4, v_7, v_6)	



GRAFOS PONDERADOS versus NO PONDERADOS

- Aplicando la misma lógica que en el caso no ponderado (versión 1):
  - se marca cada vértice como alcanzado o no alcanzado
  - se utiliza una distancia provisional por cada vértice que será la de menor coste desde el vértice inicial utilizando como intermediarios solo vértices alcanzados con la siguiente diferencia:

    - **No ponderado:**

      $d_w = d_v + 1$ si $d_w = \infty$

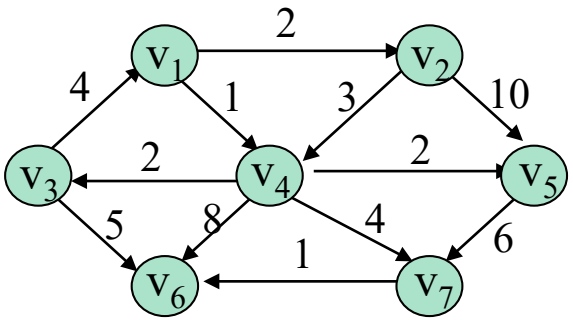
    - **Ponderado:**

      $d_w = d_v + peso(v,w)$

      si el nuevo valor de d_w ofrece una mejoría sobre el anterior

      Se actualiza d_w si $d_v + peso(v,w) < d_w$

Ejemplo: vértice origen v_1



Estado inicial

v	alc.	dist.	ant.
v_1	0	**0**	**0**
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v_1 alcanzado

v	alc.	dist.	ant.
v_1	**1**	0	0
v_2	0	**2**	**v_1**
v_3	0	∞	0
v_4	0	**1**	**v_1**
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v_4 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	**3**	**v_4**
v_4	**1**	1	v_1
v_5	0	**3**	**v_4**
v_6	0	**9**	**v_4**
v_7	0	**5**	**v_4**

v_2 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	**1**	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

v_5 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	**1**	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

v_3 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	**1**	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	**8**	**v_3**
v_7	0	5	v_4

v_7 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	**6**	**v_7**
v_7	**1**	5	v_4

v_6 alcanzado

v	alc.	dist.	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	**1**	6	v_7
v_7	1	5	v_4

Algoritmo de DIJKSTRA

```
procedimiento DIJKSTRA(vInicio:tipoIdVertice,ref  g: tipoGrafo)
1.      p: punteroArco
2.      v,w : tipoIdVértice
3.      i: entero
4.      iniciar(g)
5.      g.directorio[vInicio].distancia ← 0
6.      g.directorio[vInicio].anterior ← 0
7.       para i=1 hasta g.orden hacer
8.            v ← buscarVérticeDistanciaMínimaNoAlcanzado(g)
9.           g.directorio[v].alcanzado ←VERDADERO
10.            p ← g.directorio[v].listaAdyacencia
11.          mientras p ≠ NULO hacer
12.                 w ←p↑.vértice
13.             si g.directorio[w].alcanzado = FALSO entonces
14.               si g.directorio[v].distancia+p↑.peso<g.directorio[w].distancia
15.                  entonces
16.                   g.directorio[w].distancia ← g.directorio[v].distancia + p↑.peso
17.                        g.directorio[w].anterior ← v
18.                    fin si
19.             fin si
20.             p ← p↑.sig
21.      fin   mientras
22.   fin para
```

Análisis

- El tiempo de ejecución es $O(a+n^2)$
  - La función *buscarVérticeDistanciaMínimaNoAlcanzado* toma un tiempo $O(n)$ en recorrer el directorio. Se llama n veces por tanto consumirá en todo el algoritmo un tiempo de $O(n^2)$
  - El tiempo para actualizar d_w es constante y se ejecuta como mucho una vez por arista
- Si el grafo es denso ($a=n^2$) el algoritmo es sencillo y óptimo: se ejecuta en un tiempo lineal sobre el número de aristas
- Si el grafo es disperso ($a<<n^2$) el algoritmo de Dijkstra es demasiado lento

- MEJORA:
  - Utilización de una cola de prioridad

Estrategia

- Utilizar un montículo en el que se guarda el nuevo valor de la distancia cada vez que se ajusta un vértice (clave del montículo) y el identificador del vértice ajustado.
- Cada elemento del montículo contiene
  - clave: distancia conseguida
  - información: identificador del vértice
- Puede haber más de un representante de cada vértice en la cola de prioridad pero siempre se elimina primero la ocurrencia de distancia mínima (criterio de orden) y en ese momento se marca (no puede conseguirse una distancia mejor)
- La operación *buscarVérticeDistanciaMínimaNoAlcanzado* debe convertirse en un ciclo que ejecutará *eliminarMin* hasta que aparezca un vértice no alcanzado
  - El tamaño de la cola de prioridad puede llegar a coincidir con el número de aristas

Algoritmo de DIJKSTRA (versión 2)

```
procedimiento DIJKSTRA(vInicio:tipoIdeVertice,ref  g: tipoGrafo)
1.  p: punteroArco
2.  v,w : tipoIdVértice
3.  m: tipoMontículo
4.  x: tipoElemento     // del montículo
5.  i: entero

6.      iniciar(g)
7.      g.directorio[vInicio].distancia ← 0
8.      g.directorio[vInicio].anterior ← 0
9.      creaVacío(m )
10.     x.clave ← 0          // distancia mínima
11.     x.información ← vInicio          // identificador de vértice
12.     inserta(x,m)
13.     mientras NOT(vacio(m)) hacer
            …
32.     fin mientras
```

Algoritmo de DIJKSTRA (versión 2)

```
13. mientras NOT(VACIO(m)) hacer
14.     x ←eliminarMin(m)
15.     if  NOT(g.directorio[x.información].alcanzado) entonces
16.      v ← x.información
17.      g.directorio[v].alcanzado ←VERDADERO
18.      p ← g.directorio[v].listaAdyacencia
19.      mientras p ≠ NULO hacer
20.         w ← p↑.vértice
21.         si NOT (g.directorio[w].alcanzado ) entonces
22.            coste = g.directorio[v].distancia + p↑.peso
23.            si  coste < g.directorio[w].distancia entonces
24.               g.directorio[w].distancia ← coste
25.               g.directorio[w].anterior ← v
26.               x.clave ←g[w].distancia
27.               x.información ←w
28.               inserta(x,M)
29.            fin si
30.         fin si
31.         p ← p↑.sig
32.      fin mientras
33.   fin si
34. fin mientras
```

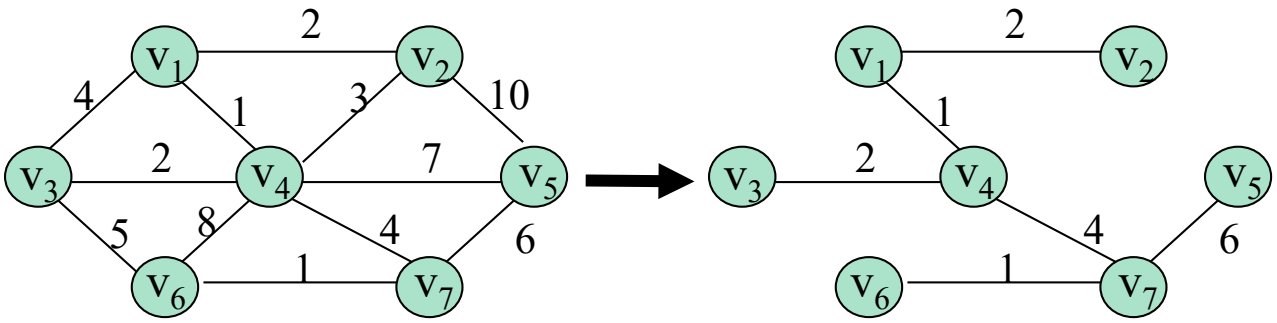
Análisis

- La operación *buscarVérticeDistanciaMínimaNoAlcanzado* debe convertirse en un ciclo que ejecutará *eliminarMin* hasta que aparezca un vértice no alcanzado
  - La operación *eliminarMin* toma un tiempos en *O(logt)* siendo *t* el tamaño del montículo
  - El tamaño del montículo puede llegar a coincidir con el número de aristas
  - El tiempo de ejecución por tanto está en *O(alogn)* ya que para grafos dispersos $a < n^2$ y por tanto $\log a < 2 \log n$

6 ÁRBOL DE EXPASIÓN DE COSTE MÍNIMO

Definición:

- Un árbol de expansión mínimo de un grafo **no** dirigido G es el árbol formado a partir de las aristas que conectan todos los vértices de G con un coste total mínimo
- Ejemplo (raíz del árbol v_1):



Algoritmos básicos que resuelven el problema: Prim y Kruskal

Dos estrategias voraces

- Algoritmo de Prim
  - Selecciona un vértice y construye el árbol a partir de ese vértice, seleccionando en cada etapa la arista más corta que extienda el árbol
- Algoritmo de Kruskal
  - Selecciona en cada paso la arista más corta que todavía no se haya considerado

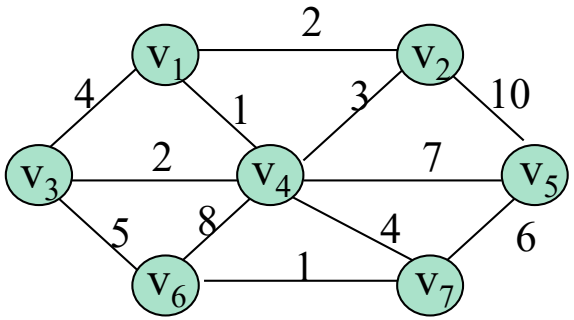
Algoritmo de PRIM

- **Proceso**: hacer crecer el árbol en etapas sucesivas, de forma que en cada etapa se agrega al árbol una arista (la de menor peso) y con ella su vértice asociado
  1. Elegir un vértice cualquiera ***u*** del grafo como nodo raíz
  2. Repetir mientras queden vértices por añadir al árbol:
     - Seleccionar la arista (u ,v) con menor peso entre todas las aristas tal que ***u*** está en el árbol y ***v*** no
     - Agregar ***v*** al árbol
- Como en el algoritmo de Dijkstra se necesita mantener información sobre cada vértice :
  - **alcanzado**: indica si el vértice ya se ha incluido en el árbol
  - **peso**: peso del arco de menor coste que conecta ese vértice con un vértice alcanzado
  - **anterior**: identificador del último vértice que ocasiona un cambio en peso

- En cada etapa se agrega al árbol un vértice ***v*** si el peso de la arista (u,v) es el menor entre todas las aristas tal que ***u*** está en el árbol y ***v*** no
- Una vez elegido ***v***, para cada vértice ***w*** no alcanzado adyacente a ***v*** se actualiza el peso y anterior, si se obtiene un peso menor:

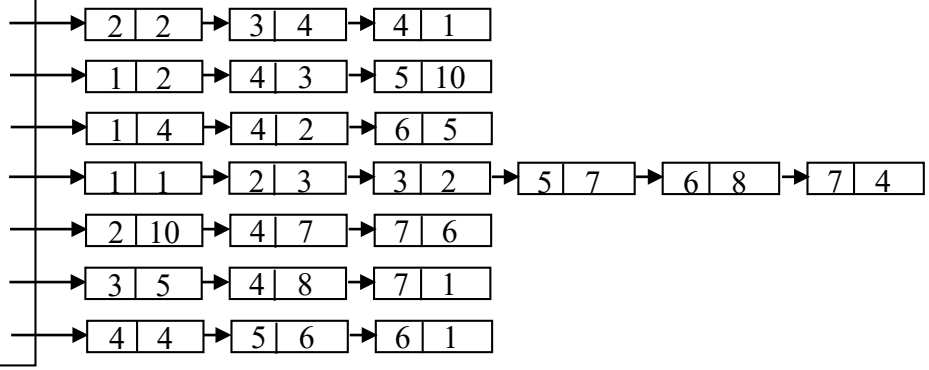
$$p_w = min(p_w\ ,\ \text{peso}(w,v))$$

Ejemplo: vértice inicial v_1



Estado inicial

v	alc.	peso	ant.
v_1	0	**0**	**0**
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0



v_1 alcanzado

v	A	P	ant.
v_1	**1**	0	0
v_2	0	**2**	**v_1**
v_3	0	**4**	**v_1**
v_4	0	**1**	**v_1**
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v_4 alcanzado

v	A	P	ant.
v_1	1	0	0
v_2	0	2	v_1
v_3	0	**2**	**v_4**
v_4	**1**	1	v_1
v_5	0	**7**	**v_4**
v_6	0	**8**	**v_4**
v_7	0	**4**	**v_4**

v_2 y v_3 alcanzados

v	A	P	ant.
v_1	1	0	0
v_2	**1**	2	v_1
v_3	**1**	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	**5**	**v_3**
v_7	0	4	v_4

v_7 alcanzado

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	**6**	**v_7**
v_6	0	**1**	**v_7**
v_7	**1**	4	v_4

v_6 y v_5 alcanzados

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	**1**	6	v_7
v_6	**1**	1	v_7
v_7	1	4	v_4

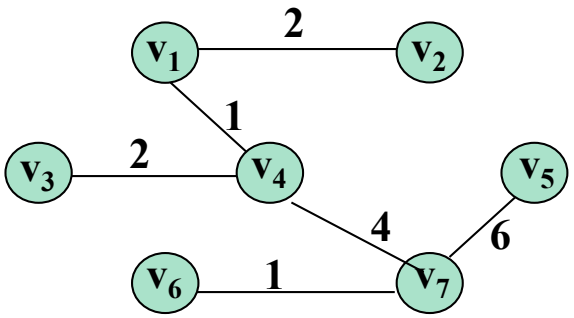
Algoritmo de PRIM

```
procedimiento PRIM(vInicio: tipoIdVértice, ref g: tipoGrafo)
1.    p : punteroArco
2.    v,w : tipoIdVértice
3.    i: entero
4.    iniciar(g)
5.    g.directorio[vInicio].peso ← 0
6.    g.directorio[vInicio].anterior ← 0
7.    para i=1 hasta g.orden hacer
8.        v ← buscarVérticeCostoMínimoNoAlcanzado(g)
9.        g.directorio[v].alcanzado ←VERDADERO
10.       p ← g.directorio[v].listaAdyacencia
11.       mientras p ≠ NULO hacer
12.          w ← p↑.vértice
13.          si not(g.directorio[w].alcanzado) entonces
14.              si g.directorio[w].peso > p↑.peso entonces
15.                      g.directorio[w].peso ← p↑.peso
16.                      g.directorio[w].anterior ← v
17.                  fin si
18.          fin si
19.          p ← p↑.sig
20.       fin mientras
21.   fin para
```

Observaciones:

- La implantación completa del algoritmo es prácticamente idéntica al algoritmo de Dijkstra
- El algoritmo de Prim se ejecuta sobre grafos **no dirigidos**: todas las aristas deben aparecer en dos listas de adyacencia
- El tiempo de ejecución es $O(n^2)$ y puede mejorarse para grafos poco densos a $O(alogn)$ utilizando montículos binarios

v	A	P	ant.
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	**1**	6	v_7
v_6	**1**	1	v_7
v_7	1	4	v_4

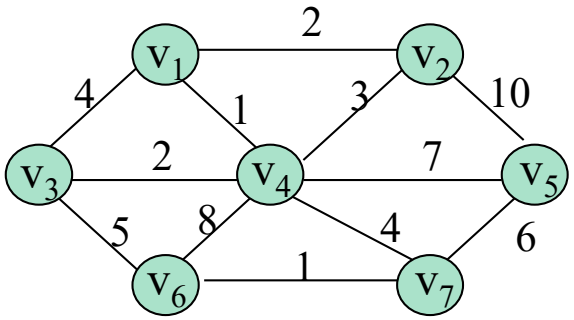


Estrategia del algoritmo de Kruskal

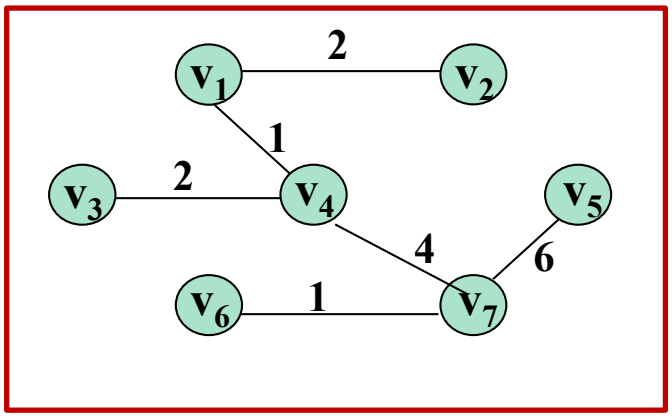
1. Crear un bosque *B* (conjunto de árboles), donde cada vértice del grafo forma un árbol diferente
2. Crear un conjunto *A* que contenga a todas las aristas del grafo
3. Repetir mientras *A* tenga aristas
   - eliminar de *A* la arista de menor peso
   - si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol, en caso contrario se rechaza

- Uso de los conjuntos disjuntos estudiados en el tema anterior con sus operaciones unión/buscar
  - Inicialmente hay n árboles de un solo nodo (crear(B))
  - Cada vez que se agrega una arista, se combinan dos árboles en uno (union(u,v,B))
  - Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo

Ejemplo



Crear un bosque *B* donde cada vértice del grafo forma un árbol diferente



Aceptar aristas si conecta dos árboles diferentes

A veces no hace falta revisar todas las aristas

Crear un conjunto *A* que contenga a todas las aristas del grafo

Arista	Peso	Aceptada
(v_1,v_4)	1	**SI**
(v_6,v_7)	1	**SI**
(v_1,v_2)	2	**SI**
(v_3,v_4)	2	**SI**
(v_2,v_4)	3	**NO**
(v_1,v_3)	4	**NO**
(v_4,v_7)	4	**SI**
(v_3,v_6)	5	**NO**
(v_5,v_7)	6	**SI**
(v_4,v_5)	7	**NO**
(v_4,v_6)	8	**NO**
(v_2,v_5)	10	**NO**

Utilización de conjuntos disjuntos

- En cualquier momento del proceso, dos vértices pertenecen al mismo conjunto si y sólo si, están conectados en el bosque de expansión actual
  - Relación de equivalencia: "estar conectado"
  - Inicialmente, cada vértice en su propio conjunto (crea): ningún vértice conectado
  - Según se aceptan aristas, se van conectando vértices. Cada vez que se conectan dos vértices (unir) quedan en el mismo conjunto
  - Solo se acepta una nueva arista *(u,v)*, si *u* y *v* no están en el mismo conjunto (buscar(u)<> buscar(v)
  - Si los vértices que conecta esa arista están en el mismo conjunto, significa que ya están conectados, añadirla crearía un ciclo
  - Cada vez que se agrega una arista *(u,v)* al bosque de extensión los vértices del conjunto *u* quedan conectados con los vértices del conjunto *v*
  - Si *x* esta conectado con *u* y *w* esta conectado con *v*. Agregar la arista *(u,v)* significa que *x* y *w* pasan a estar conectados

Algortimo de Kruskal

```
función Kruskal(ref g: tipoGrafo):tipoGrafo
1.  m: tipoMontículo
2.  numAristasAceptadas: entero
3.  C: tipoPartición
4.  conjuntoU, conjuntoV: tipoConjunto
5.  x: tipoElemento   //  aristas del grafo (u, v)
6.  arbolExp: tipoGrafo
7.  crear(C)
8.  construirMontículoDeAristas(g,m)
9.  numAristasAceptadas ← 0
10. mientras numAristasAceptadas < g.orden -1 hacer
11.         x ← eliminarMin(m)
12.         conjuntoU ← buscar(x.información.u,C)
13.         conjuntoV ← buscar(x.información.v,C)
14.         si conjuntoU != conjuntoV entonces
15.                 unir(C, conjuntoU, conjuntoV)
16.                 numAristasAceptadas ← numAristasAceptadas + 1
17.                 aceptarArista (x, arbolExp)
18.         fin si
19. fin mientras
20. devolver arbolExp
```

Análisis

- El algortimo de Kruskal se puede implantar para que se ejecute en un tiempo *O(alogn)*
  - Si el grafo tiene *a* aristas, construir el montículo de aristas lleva un tiempo en *O(aloga)*, que puede mejorarse a *O(a)*
  - En el peor de los casos, que sea necesario revisar todas las aristas del grafo, las operaciones en la cola de prioridad llevan un tiempo *O(aloga)*. Normalmente no es necesario revisarlas todas para obtener el árbol de expansión mínimo
  - El tiempo total requerido por las operaciones buscar/unir en la estructura partición, depende del método utilizado en su implantación, sabemos que hay métodos en *O(aloga)* y *O(aα(a))*
  - El tiempo de ejecución por tanto está en *O(alogn)* ya que para grafos dispersos $a < n^2$ y por tanto $loga < 2\ logn$

Ejercicio 1

Razonar brevemente que información nos aporta la siguiente tabla, teniendo en cuenta que representa parte del directorio de vértices de un grafo después de aplicar el algoritmo de DIJKSTRA. El razonamiento implica establecer los caminos mínimos obtenidos: el coste de cada camino y la secuencia de vértice que forman cada camino

En la tabla sólo se representa la información del directorio de vértices relevante para el algoritmo de DIJKSTRA: identificador de vértice, distancia y anterior

Vértice	distancia	anterior
1	0	0
2	5	4
3	3	1
4	4	3
5	6	4

Ejercicio 2

Dada la representación en memoria que se muestra en la siguiente figura y que se corresponde con un grafo después de aplicar el algoritmo de PRIM. Donde:

- sólo se representa la información del directorio de vértices relevante para el algoritmo de Prim: identificador de vértice (V), coste o peso (P) y anterior (A).
- las celdas de las listas de adyacencia tienen dos campos de información el primero sobre el vértice adyacente y el segundo sobre el peso del arco

Teniendo en cuenta el contenido de esta estructura de datos, se pide:

a) Dibujar el grafo

b) Dibujar el árbol de expansión

Figura ejercicio 2

V	C	A													
1	2	4	→	2	5	→	3	4	→	4	2				
2	1	4	→	1	5	→	4	1	→	5	17				
3	3	4	→	1	4	→	4	3	→	6	5				
4	7	5	→	1	2	→	2	1	→	3	3	→	5	7	→	6	6	→	7	1
5	0	0	→	2	17	→	4	7	→	7	15				
6	5	3	→	3	5	→	4	6	→	7	10				
7	1	4	→	4	1	→	5	15	→	6	10				

Aplicación RAED

La aplicación RAED permite estudiar y analizar los diferentes algoritmos aplicados a distintos grafos ejemplo.

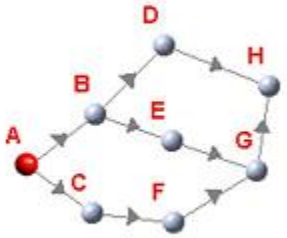


Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	false	1	0
C	false	1	0
D	false	@	-1
E	false	@	-1
F	false	@	-1
H	false	@	-1
G	false	@	-1

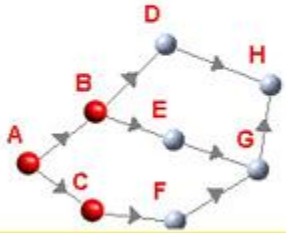


Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	true	1	0
C	true	1	0
D	false	2	1
E	false	2	1
F	false	2	2
H	false	@	-1
G	false	@	-1

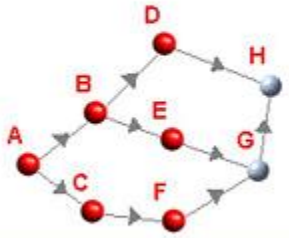


Tabla de vértices :

VERTICE	ALCANZADO	DISTANCIA	ANTERIOR
A	true	0	-1
B	true	1	0
C	true	1	0
D	true	2	1
E	true	2	1
F	true	2	2
H	false	3	3
G	false	3	4

Ejercicio 3

a) Proponer un algoritmo en pseudocódigo que determine si un grafo es conexo mediante clases de equivalencia

b) Implementar el algoritmo en C

función conexo(**ref** g: tipoGrafo) **: lógico**

Idea básica.

- Si dos vértices del grafo están conectados pertenecen a la misma clase de equivalencia, por tanto:
  - Si todos los vértices de un grafo están en la misma clase de equivalencia (están conectados) el grafo es **conexo**
  - Si hay varias clases de equivalencia el grafo **no** es **conexo**