Tema 15 - Soluciones Greedy para Problemas de Optimización sobre un Grafo

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tema 15 - Soluciones Greedy para Problemas de Optimización sobre un Grafo

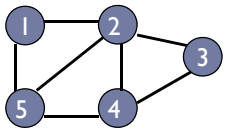
Índice general:

Bibliografía

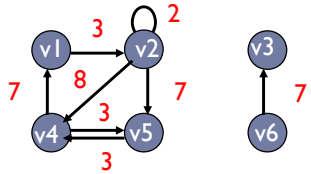
- Libro de M.A. Weiss, "Estructuras de Datos en Java" (Adisson-Wesley, 2000).
  - Capítulo 22, apartado 22.2.3 para el algoritmo de Dijkstra con Montículos de Emparejamiento

Caminos Mínimos en un Grafo

- El problema del **cálculo de los caminos mínimos** en un Grafo es de gran interés dado que muchas aplicaciones reales pueden modelarse mediante esta estructura de datos.
  - Obtención de la ruta más rápida entre dos puntos.
  - Gestión de paquetes en una red de computadores.
- Variantes del problema:
  - Cálculo del camino mínimo en Grafos **sin pesos** y en Grafos **con pesos**. El camino mínimo en un grafo ponderado NO tiene porqué ser el camino más corto.



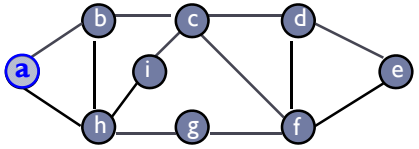
Grafo sin pesos:
Camino mínimo en nº de aristas



Grafo con pesos:
Camino mínimo considerando pesos

Problema del Camino Mínimo Sin Pesos con Único Origen (I)

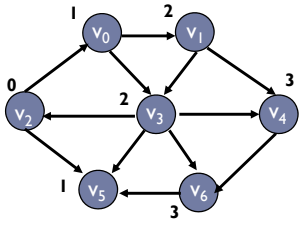
- Encontrar el camino más corto (medido en n° de aristas) desde un vértice origen a cualquier otro vértice.
  - El problema del camino sin pesos es un caso particular del camino con pesos donde todos los pesos valen 1.



- Vértice Origen: a
  - Nodos a distancia 0 de a → {a}
  - Nodos a distancia 1 de a → {b, h} (adyacentes a los de dist. 0)
  - Nodos a distancia 2 de a → {c, i, g} (adyacentes a los de distancia 1)

Problema del Camino Mínimo Sin Pesos con Único Origen (II)

- Para obtener los vértices cuyo camino mínimo desde el vértice origen tiene longitud *k* nos apoyamos en los nodos adyacentes cuyo camino desde el vértice origen es *k-1*.
- En realidad estamos realizando un recorrido en anchura, donde los vértices más cercanos al vértice origen se procesan antes que los más alejados, trabajando por niveles.



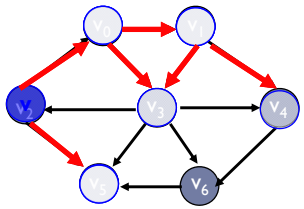
- Vértice Origen: V_2
- Nodos a distancia 0 de V_2 → $\{V_2\}$
- Nodos a distancia 1 de V_2 → $\{V_0, V_5\}$
- Nodos a distancia 2 de V_2 → $\{V_1, V_3\}$
- Nodos a distancia 3 de V_2 → $\{V_4, V_6\}$

Reformulación del BFS para el Cálculo del Camino Mínimo Sin Pesos

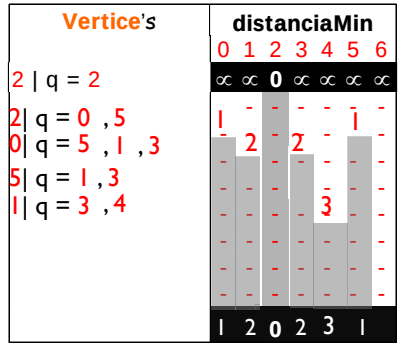
- Recorrido en Anchura o Breadth First Search (BFS) del Vértice dado, vOrigen, en el que *visitados* pasa a ser ***distanciaMin***.
- distanciaMin[i] representa la Distancia mínima desde el Vértice dado vOrigen al Vértice i.
  - distanciaMin[vOrigen] == 0
  - si distanciaMin[i] == ∞ el Vértice i no se ha visitado aún
  - si distanciaMin[i] != ∞ el Vértice i sí se ha visitado, desde un cierto vértice vAnterior: distanciaMin[i]=distanciaMin[vAnterior]+1

Traza de Camino Mínimo Sin Pesos (distanciaMin)

- Vértice origen: V_2



como distanciaMin**[3] != ∞**
NO cambia su valor a **3**

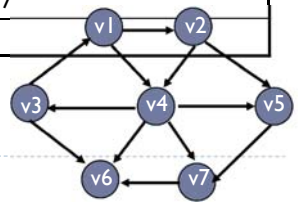


- De esta manera el algoritmo controla los vértices que ya han sido visitados anteriormente.

Aunque en esta traza los vértices se numeren de 0..N, de normal los numeramos de 1..N

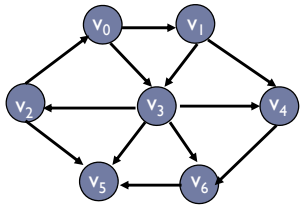
Traza de Camino Mínimo Sin Pesos (distanciaMin): Tabla Final

Vértice	distanciaMin	Cola
	1 2 3 4 5 6 7 ∞ ∞ ∞ ∞ ∞ ∞ ∞	
	∞ ∞ 0 ∞ ∞ ∞ ∞	3
3: 1 6	1 ∞ 0 ∞ ∞ 1 ∞	1 6
1: 2 4	1 2 0 2 ∞ 1 ∞	6 2 4
6:	1 2 0 2 ∞ 1 ∞	2 4
2: 4 5	1 2 0 2 3 1 ∞	4 5
4: 3 5 6 7	1 2 0 2 3 1 3	5 7
5: 7	1 2 0 2 3 1 3	7
7: 6	1 2 0 2 3 1 3	



Traza de Camino Mínimo Sin Pesos (caminoMin)

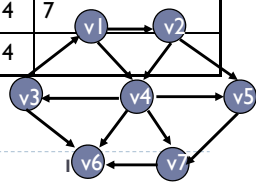
- Para guardar el camino, utilizamos un vector llamado *caminoMin:*
  - caminoMin[i] == -1 si el Vértice i aún NO es del Camino Mínimo.
  - caminoMin[i] == vAnterior si el Vértice i se alcanza desde vAnterior en el Camino Mínimo.



Vertice's	distanciaMin	caminoMin
	0 1 2 3 4 5 6	0 1 2 3 4 5 6
2 | q =2	∞ ∞ 0 ∞ ∞ ∞ ∞	-1 -1 -1 -1 -1 -1 -1
2 | q =0	1 - - - - - -	2 - - - - - -
2 | q =0, 5	- - - - - 1 -	- - - - - 2 -
0 | q =5, 1	- 2 - - - - -	- 0 - - - - -
0 | q =5, 1, 3	- - - 2 - - -	- - - 0 - - -
5 | q =1, 3	- - - - - - -	- - - - - - -
1 | q =3, 4	- - - - 3 - -	- - - - 1 - -
3 | q =4, 6	- - - - - - 3	- - - - - - 3
4 | q =6	- - - - - - -	- - - - - - -
6 | q = ∅	- - - - - - -	- - - - - - -
	1 2 0 2 3 1 3	2 0 -1 0 1 2 3

Traza de Camino Mínimo Sin Pesos (caminoMin): Tabla Final

Vértice	distanciaMin	caminoMin	Cola
	1 2 3 4 5 6 7	1 2 3 4 5 6 7	
	∞ ∞ 0 ∞ ∞ ∞ ∞	-1 -1 -1 -1 -1 -1 -1	
3: 1 6	1 ∞ 0 ∞ ∞ 1 ∞	3 -1 -1 -1 -1 3 -1	1 6
1: 2 4	1 2 0 2 ∞ 1 ∞	3 1 -1 1 -1 3 -1	6 2 4
6:	1 2 0 2 ∞ 1 ∞	3 1 -1 1 -1 3 -1	2 4
2: 4 5	1 2 0 2 3 1 ∞	3 1 -1 1 2 3 -1	4 5
4: 3 5 6 7	1 2 0 2 3 1 3	3 1 -1 1 2 3 4	5 7
5: 7	1 2 0 2 3 1 3	3 1 -1 1 2 3 4	7
7: 6	1 2 0 2 3 1 3	3 1 -1 1 2 3 4	



Ampliación de la clase Grafo para Caminos Mínimos sin Pesos (I)

```
package librerias.estructurasDeDatos.grafos;
public abstract class Grafo<E> {
   …
   protected static final int INFINITO = (Integer.MAX_VALUE) / 3;
   protected int distanciaMin[];
   protected int caminoMin[];

public void caminosMinimoSinPesos(int vOrigen) {
      distanciaMin = new double[numVertices() + 1];
      caminoMin = new int[numVertices() + 1];
      for (int i = 1; i <= numVertices(); i++) {
         distanciaMin[i] = INFINITO;
         caminoMin[i] = -1;
      }
   …
```

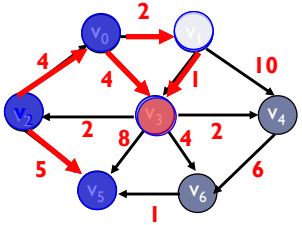
Ampliación de la clase Grafo para Caminos Mínimos sin Pesos (II)

```
...
    q = new ArrayCola<Integer>();
    q.encolar(new Integer(vOrigen));  distanciaMin[vOrigen] = 0;
    while (!q.esVacia()) {
       int v = q.desencolar().intValue();
       ListaConPI<Adyacente> aux = adyacentesDe(v);
       aux.inicio();
       while (!aux.esFin()) {
          int w = aux.recuperar().destino;
          if (distanciaMin[w] == INFINITO) { // w no visitado aún
             distanciaMin[w] = distanciaMin[v] + 1;
             caminoMin[w] = v;  q.encolar(new Integer(w));
          }
          aux.siguiente();  } } }
```


Camino Mínimo con Pesos

- **Problema de los Caminos Mínimos con Coste Positivo y Orígen Único:**
  - Encontrar el camino de longitud más corta (considerando los pesos) desde el vértice origen al resto de vértices.
  - El coste de cada arista **debe ser positivo**.
- La longitud de un camino con pesos es la suma del coste de las aristas del camino.
- El método del cálculo del camino mínimo en un Grafo sin pesos NO es directamente aplicable a este problema.

Problema al Usar Algoritmo de Camino Mínimo Sin Pesos a un Grafo Ponderado



¡distanciaMin[3] == 8 NO cambia a 7, porque solo comprueba ≠INFINITO!

Vertice's	distanciaMin						
	0	1	2	3	4	5	6
	∞	∞	**0**	∞	∞	∞	∞
2 | q = 2	4	-	-	-	-	-	-
2 | q = 0	-	-	-	-	-	5	-
2 | q = 0, 5	-	6	-	-	-	-	-
0 | q = 5, 1	-	-	-	**8**	-	-	-
0 | q = 5, 1, 3	-	-	-	-	-	-	-
5 | q = 1, 3	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
1 |q = 3	-	-	-	-	-	-	-
	4	**6**	**0**	**8**	-	**5**	-

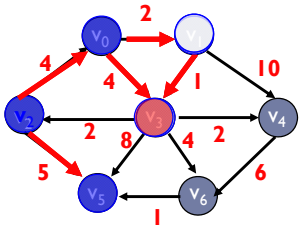
- El algoritmo de camino mínimo sin pesos sólo actualiza el contenido de distanciaMin si distanciaMin[i] == INFINITO, no considera el hecho de haber encontrado un camino más corto (considerando pesos).

Adaptación del Recorrido de la Lista de Adyacentes a un Vértice

```
while ( !aux.esFin() )  {
        Adyacente a  = aux.recuperar();
        int w = a.destino, costeW = a.peso;
        if ( distanciaMin[w] > distanciaMin[v] + costeW ) ) {
            distanciaMin[w] = distanciaMin[v] + costeW;
            caminoMin[w] = v;   q.encolar(new Integer(w));
        }
        aux.siguiente();
}
```

- Ahora se comprueba si se ha encontrado un camino de mejor coste y se actualizan los vectores distanciaMin y caminoMin.
- Problema: Es posible que en la cola introduzcamos vértices ya visitados!

Problema Adicional en la Gestión de la Cola



!!! **NO** se puede usar una simple Cola de Integer !!!

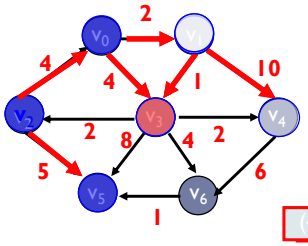
Vertice's	distanciaMin 0	1	2	3	4	5	6
	∝	∝	0	∝	∝	∝	∝
2 | q = 2	4	-	-	-	-	-	-
2 | q = 0	-	-	-	-	-	5	-
2 | q = 0, 5	-	6	-	-	-	-	-
0 | q = 5, 1	-	-	-	7	-	-	-
0 | q = 5, 1, 3	-	-	-	-	-	-	-
5 | q = 1, 3	-	-	-	-	-	-	-
1 | q = 3 , 3	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	4	6	0	7	-	5	-

- Para evitar introducir en la Cola elementos repetidos, tendremos que diferenciar entre haber llegado a un adyacente por una arista o por otra.
  - La diferencia se establecerá en base al coste de la arista.

Reformulación de la Cola Empleada

- Los datos de la Cola ya no serán Integer, sino **Pares** de Integer (adyacente, coste global de llegar a él desde el vértice origen). En el ejemplo, se encolarían: (2,0), (0,4), (5,5), (1,6), (3,8), (3,7).
- Cambios a realizar:
  - Añadir la clase DistanciaHastaV (que representa un par) a la estructura de clases actual.
  - q.encolar(new Integer(vOrigen))→q.encolar(new DistanciaHastaV (vOrigen,0))
  - q.encolar(new Integer(w)) → q.encolar(new DistanciaHastaV (w, distanciaMin[w]))
- Los Pares de Integer se ordenan por Distancia, por lo que la Cola pasa a ser una Cola de Prioridad, implementada como un Heap. En el ejemplo, (3,7) se colocará ANTES de (3,8).

Uso de Cola de Prioridad en Problema de Caminos Mínimos con Pesos



(4,16)

Vertice's	distanciaMin 0	1	2	3	4	5	6
	∝	∝	0	∝	∝	∝	∝
2|qP=(2,0)	4	-	-	-	-	-	-
2|qP=(0,4)	-	-	-	-	-	5	-
2|qP=(0,4),(5,5)	-	6	-	-	-	-	-
0|qP=(5,5),(1,6)	-	-	-	7	-	-	-
0|qP=(5,5),(1,6),(3,8)	-	-	-	-	-	-	-
5|qP=(1,6),(3,8)	-	-	-	-	-	-	-
1|**qP=(3,7)** ,(3,8)	-	-	-	-	-	-	-
1|**qP**=(3,7),(3,8),(4,16)	-	-	-	-	9	-	-
3|qP=(3,8),(4,9),(4,16)							11
3|qP=(3,8),(4,10),(4,16),(6,11)	-	-	-	-	-	-	-
	4	6	0	7	9	5	-

- Hay que marcar los Vértices ya desencolados para no volverlos a procesar: (3,7), (3,8).

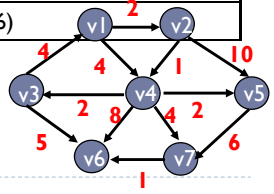
Gestión de Vértices Repetidos

- Hay que marcar los Vértices ya visitados para no volverlos a procesar :
  - Sin este mecanismo, en el ejemplo anterior procesaríamos el vértice 3 dos veces: (3,7), (3,8).
- Únicamente interesa procesar el par por el cual se incurre en un coste menor y eso lo controla la Cola de Prioridad.
- Este mecanismo nos permite evitar procesar vértices ya procesados anteriormente.
- Utilizamos un atributo para marcar los Vértices ya visitados:
  - si visitados[i] == 0, el Vértice i NO se ha visitado aún
  - si visitados[i] == 1, el Vértice i SÍ se ha visitado

Traza Completa de Camino Mínimo con Cola de Prioridad

Vértice	distanciaMin	visitados	Cola de Prioridad
	1 2 3 4 5 6 7 ∞ ∞ 0 ∞ ∞ ∞ ∞	1 2 3 4 5 6 7 0 0 0 0 0 0 0	(3,0)
3: 1 6	4 ∞ 0 ∞ ∞ 5 ∞	0 0 1 0 0 0 0	(1,4), (6,5)
1: 2 4	4 6 0 8 ∞ 5 ∞	1 0 1 0 0 0 0	(6,5), (2,6), (4,8)
6:	4 6 0 8 ∞ 5 ∞	1 0 1 0 0 1 0	(2,6), (4,8)
2: 4 5	4 6 0 7 16 5 ∞	1 1 1 0 0 1 0	(4,7), (4,8), (5,16)
4: 3 5 6 7	4 6 0 7 9 5 11	1 1 1 1 0 1 0	(4,8), (5,9), (7,11), (5,16)
5: 7	4 6 0 7 9 5 11	1 1 1 1 1 1 0	(7,11), (5,16)
7: 6	4 6 0 7 9 5 11	1 1 1 1 1 1 1	(5,16)

- Faltaría incorporar la información de los caminos mínimos



Ampliación de la Clase Grafo para Caminos Mínimos con Pesos

```
public abstract class Grafo<E> {
   …
   protected ColaPrioridad<DistanciaHastaV> qPrior;

   public void dijkstra(int vOrigen) { … }

} /* Fin de la clase Grafo<E> */
```


El método dijkstra de CaminosDelGrafo (1/2)

```
public void dijkstra(int vOrigen) {
      distanciaMin = new double[numVertices() + 1];
      caminoMin = new int[numVertices() + 1];
      visitados = new int[numVertices() + 1];
      for (int i = 1; i <= numVertices(); i++) {
         distanciaMin[i] = INFINITO;
         caminoMin[i] = -1;
      }
      distanciaMin[vOrigen] = 0;
      ColaPrioridad<DistanciaHastaV> qPrior = new
           MonticuloBinario<DistanciaHastaV>();
      qPrior.insertar(new DistanciaHastaV(vOrigen, 0));
…
```


El método dijkstra de CaminosDelGrafo (2/2)

```
while (!qPrior.esVacia()) { // Mientras haya vértices por explorar
        // El siguiente vértice a explorar es el de menor distancia
        int v = qPrior.eliminarMin().codigo;
        if (visitados[v] == 0) {        // Evitamos repeticiones
           visitados[v] = 1;
          ListaConPI<Adyacente> aux = adyacentesDe(v);
           for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
              Adyacente a = aux.recuperar();
              int  w = a.destino;   double costeW = a.peso;
              // Vemos si la mejor forma de alcanzar w  es a través de v
              if (distanciaMin[w] > distanciaMin[v] + costeW) {
                 distanciaMin[w] = distanciaMin[v] + costeW;
                 caminoMin[w] = v;
                 qPrior.insertar(new DistanciaHastaV(w, distanciaMin[w]));
              } } } } }
```

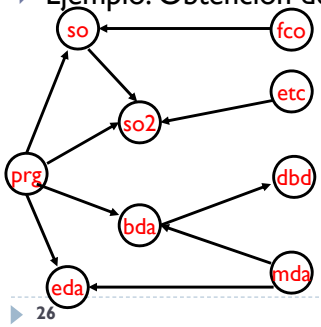
$T_{dijkstra} \in O(|E| \log |E|)$

La Clase DistanciaHastaV

```
package librerias.estructurasDeDatos.grafos;
public class DistanciaHastaV implements
   Comparable<DistanciaHastaV> {
   protected int codigo;
   protected int coste;
   public DistanciaHastaV(int codigo, int coste){
         this.codigo = codigo;
         this.coste = coste;
   }
   public int compareTo(DistanciaHastaV eh){
         if (coste < eh.coste) return -1;
         else if (coste > eh.coste) return 1;
              else return 0;
   }}
```


Ordenación Topológica

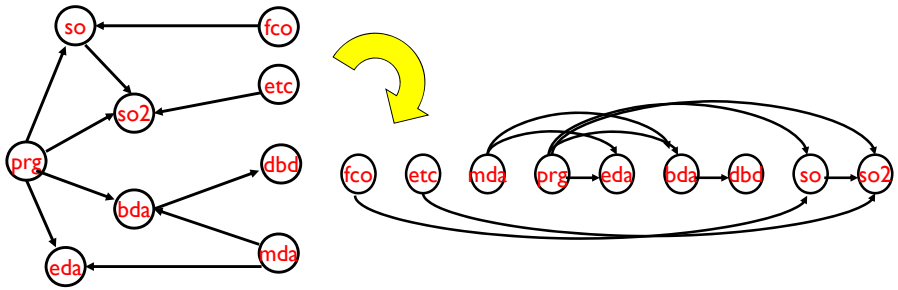
- Un **Orden Topológico** ordena los vértices de un grafo dirigido **acíclico** de manera que si hay un camino de *u* a *v* entonces *u* aparece antes que *v* en la ordenación.
- No se puede obtener un orden topológico en un grafo con ciclos.
- Ejemplo: Obtención de prerrequisitos de asignaturas.



- Ejemplos de ordenes topológicos:
  - <fco,so,so2>
  - <prg,bda,dbd>
  - <prg,eda>
- Una **Ordenación Topológica** encuentra todos los ordenes topológicos sobre un grafo **acíclico** dado.

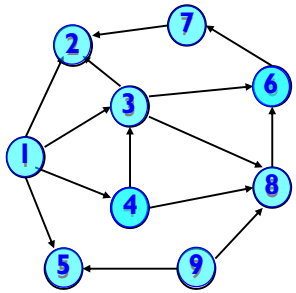
Estrategia para el Cálculo de la Ordenación Topológica

- Empleamos un recorrido en profundidad (DFS) pero utilizando una **Pila** para almacenar aquellos elementos que ya han sido **recorridos completamente**.
- El uso de una Pila permite obtener los caminos en el orden topológico correcto.



Ejemplo de Ordenación Topológica

- Aplicamos la Ordenación Topológica al siguiente grafo:
  - El método nos devolverá una secuencia de vértices que cumplen una ordenación topológica.
  - Para ello, realizaremos una traza del método de ordenación topológica para ver como se va llenando la pila de elementos ya visitados.
  - Partimos del vértice 1

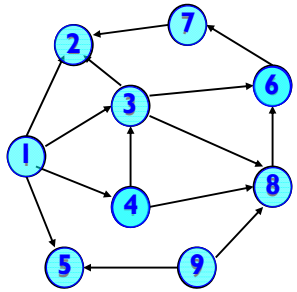


Ejemplo de Ordenación Topológica

Vértice	Visitados	Pila pVrecorridos
	1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0	
1: 2 3 4 5	1 0 0 0 0 0 0 0 0	<>
2: {}	1 2 0 0 0 0 0 0 0	<2>
3: 2 6 8	1 2 3 0 0 0 0 0 0	<2>
6: 7	1 2 3 0 0 4 0 0 0	<2>
7: 2	1 2 3 0 0 4 5 0 0	<7, 2>
---	1 2 3 0 0 4 5 0 0	<6,7,2>
8: 6	1 2 3 0 0 4 5 6 0	<8,6,7,2>
---	1 2 3 0 0 4 5 6 0	<3,8,6,7,2>
4: 3 8	1 2 3 7 0 4 5 6 0	<4,3,8,6,7,2>
5: {}	1 2 3 7 8 4 5 6 0	<5,4,3,8,6,7,2>
---	1 2 3 7 8 4 5 6 0	<1,5,4,3,8,6,7,2>
9: 5 8	1 2 3 7 8 4 5 6 9	<9,1,5,4,3,8,6,7,2>

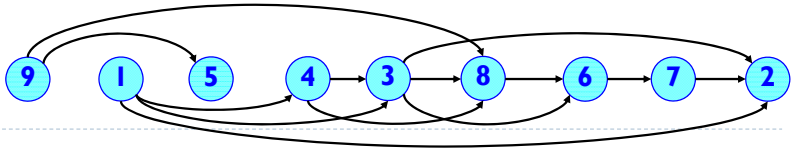
Ejemplo de Ordenación Topológica

- Si representamos la secuencia de vértices obtenida en línea y dibujamos las aristas, se puede apreciar mejor el orden topológico.



Grafo Original

Grafo mostrado en orden topológico



Modificaciones en la clase Grafo para Ordenación Topológica

```
public abstract class Grafo<E> {
   …
   /* Para la ordenación topológica */
   protected Pila<Integer> pVRecorridos;

 public int[] toArrayTopologico() { … }
  protected void ordenacionTopologica(int origen, Pila<Integer>
    pVRecorridos) {… }

}
```


Implementación de Ordenación Topológica (1)

```
public int[] toArrayTopologico() {
      visitados = new int[numVertices() + 1];
      ordenVisita = 1 ;
      Pila<Integer> pVRecorridos = new ArrayPila<Integer>();
      for (int vOrigen = 1; vOrigen <= numVertices(); vOrigen++) {
         if (visitados[vOrigen] == 0)
            ordenacionTopologica(vOrigen, pVRecorridos);
      }
      int res[] = new int[numVertices() + 1];
     for (int i = 1; i <= numVertices(); i++) res[i] = pVRecorridos.desapilar();
      return res;
   }
```

Implementación de Ordenación Topológica (2)

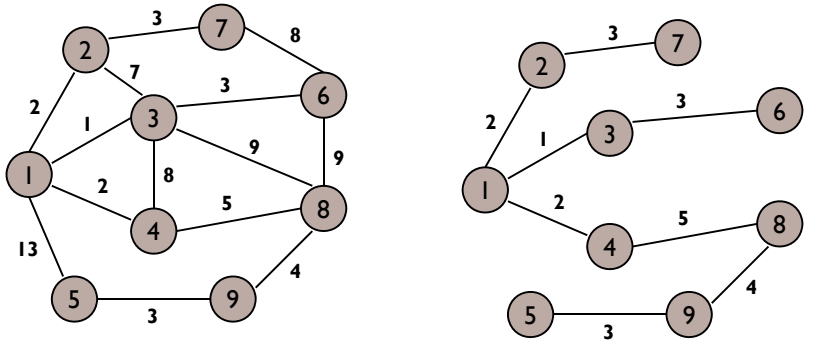
```
protected void ordenacionTopologica(int origen,
   Pila<Integer> pVRecorridos) {
      visitados[origen] = ordenVisita++;
      ListaConPI<Adyacente> aux = adyacentesDe(origen);
      for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
         int destino = aux.recuperar().destino;
         if (visitados[destino] == 0)
            ordenacionTopologica(destino, pVRecorridos);
      }
      pVRecorridos.apilar(new Integer(origen));
   }
```


Árbol Generador Mínimo: Introducción

- Un grafo no dirigido es **conexo** si cualquier par de vértices están conectados por un camino.
- Un árbol es un grafo no dirigido acíclico y conexo.
- Un árbol generador mínimo (o árbol recubridor mínimo (minimum spanning tree)) de un grafo es un subgrafo que es un árbol y permite conectar todos los vértices donde la suma del peso de sus aristas debe ser lo más pequeña posible.
  - El árbol generador mínimo de un grafo tiene:
    - El mismo número de vértices
    - Un subconjunto de sus aristas
- Dos algoritmos de cálculo del árbol generador mínimo.
  - Algoritmo de Kruskal
  - Algoritmo de Prim

Árbol Generador Mínimo: Ejemplo

- Ejemplo de árbol generador mínimo a partir de un grafo.
  - Cableado de una red de datos en un área metropolitana



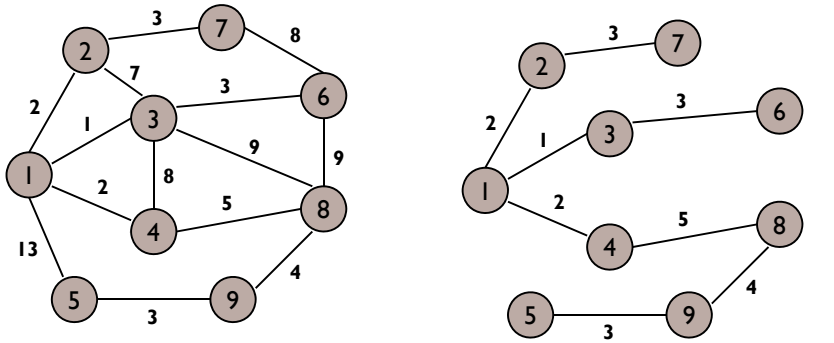
- El árbol generador mínimo permite conectar todos los vértices con coste mínimo.

Algoritmo de Kruskal

1. Guardar las aristas en una Cola de Prioridad en base a la ponderación de la arista.
2. Partimos de un grafo sin aristas (solo con vértices).
3. Mientras |E|<|V|-1 hacer:
   1. Recuperar y eliminar la arista de menor coste de la cola de prioridad.
   2. Si no provoca ciclos en el grafo, incluir la arista en el grafo.

- Consideraciones:
  - Nótese que para conectar un grafo con N vértices son necesarias N-1 aristas.
  - El resultado final es un árbol (grafo acíclico).
- Complejidad temporal:
  - $T_{kruskal}(|E|, |V|) = \mathbf{O}(|E| * \log_2(|V|))$

Ejemplo Animado del Algoritmo de Kruskal



- Se van eligiendo las aristas de menor de coste y, si no forman ciclos, se añaden al árbol.
  - A igualdad de coste, las aristas se escogen de forma arbitraria.

Algoritmo de Prim

- El algoritmo de Kruskal siempre selecciona la arista de menor peso, causando una construcción desordenada del árbol.
- El algoritmo de Prim provoca un crecimiento ordenado a partir de un determinado vértice (elegido de forma arbitraria).
- Pasos:

1. El árbol resultante (V', E') contiene inicialmente un solo vértice (arbitrario) y ninguna arista.
2. Mientras |V| < |V'| hacer
   1. Buscar la arista e = (u,v) de coste mínimo, con u ∈ V' y v no pertenece a V'.
   2. E' = E' + e
   3. V' = V' + v

- Complejidad temporal: $T_{prim}(|E|, |V|) = \mathbf{O}(|E| * \log_2(|V|))$