Ejercicios de clase

TEMA 6 – Grafos y Estructuras de Partición

EJERCICIO 1

Sea $G = (V, A)$ un grafo dirigido con pesos:

$V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$

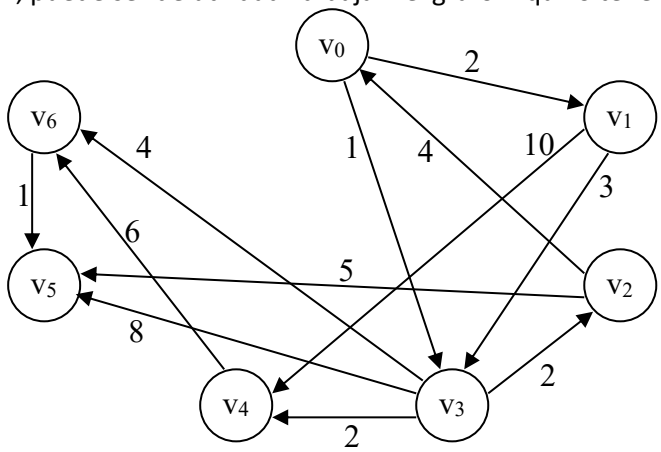
$A = \{(v_0,v_1, 2), (v_0,v_3, 1), (v_1,v_3, 3), (v_1,v_4, 10), (v_3,v_4, 2), (v_3,v_6, 4), (v_3,v_5, 8), (v_3,v_2, 2), (v_2,v_0, 4), (v_2,v_5, 5), (v_4,v_6, 6), (v_6,v_5, 1)\}$

Se pide:

a) $|V|$ y $|A|$
b) Vértices adyacentes a cada uno de los vértices
c) Grado de cada vértice y del grafo
d) Caminos desde v_0 al resto de vértices, su longitud con y sin pesos
e) Vértices alcanzables desde v_0
f) Caminos mínimos desde v_0 al resto de vértices
g) ¿Tiene ciclos?

Solución

Este ejercicio ya lo hemos planteado en clase y nos va a servir para repasar todos los conceptos vistos sobre grafos. En primer lugar, puede ser de utilidad "dibujar" el grafo. Aquí lo tenemos:



a) $|V| = 7$
$|A| = 12$

b) Vértices adyacentes a cada uno de los vértices:

Adyacentes(v_0) = {v_1, v_3}
Adyacentes(v_1) = {v_3, v_4}
Adyacentes(v_2) = {v_0, v_5}
Adyacentes(v_3) = {v_2, v_4, v_5, v_6}
Adyacentes(v_4) = {v_6}
Adyacentes(v_5) = ∅
Adyacentes(v_6) = {v_5}

c) Recordemos que el grado de un vértice de un grafo dirigido es la suma del grado de entrada (número de aristas que entran en el vértice) y el grado de salida (número de aristas que salen del vértice). Así, por ejemplo, el grado de salida del vértice v_0 es 2 (salen 2 aristas) y el grado de entrada es 1. Por tanto, el grado del vértice 0 es 3. Si nos fijamos, el grado del vértice v_1 también es 3, así como el de los vértices v_2, v_4, v_5 y v_6. 6. El grado del vértice v_3 es 6. Y como es el máximo, ese es el grado del grafo, 6:

Grado(v_0) = Grado(v_1) = Grado(v_2) = Grado(v_4) = Grado(v_5) = Grado(v_6) = 3
Grado(v_3) = Grado del grafo = 6

d) En cuanto a los caminos simples de 0 a 6, podemos encontrar: 0-3-4-6. Este camino tiene talla 3 (3 aristas, 4 vértices) y su peso es la suma de los pesos de las aristas que lo forman: de 0 a 3, peso 1+2+6=9. Hay otros caminos simples: 0-1-4-6 o 0-1-3-4-6… Aquí podemos verlos todos. Caminos simples desde *v0* a *v6*, y su longitud con y sin pesos:

Caminos (simples)	Longitud	Longitud con pesos
$\langle v_0, v_1, v_3, v_4, v_6\rangle$, $\langle v_0, v_1, v_3, v_6\rangle$, $\langle v_0, v_1, v_4, v_6\rangle$, $\langle v_0, v_3, v_4, v_6\rangle$, $\langle v_0, v_3, v_6\rangle$	4, 3, 3, 3, 2	13, 9 18, 9, 5

e) Desde el vértice v_0 se pueden alcanzar los vértices siguientes: en primer lugar, los adyacentes, alcanzables con 1 arista: el vértice v_1 y el v_3; también se puede alcanzar el v_4 (por ejemplo, con el camino v_0, v_1, v_4) o el v_6 (v_0, v_3, v_4, v_6), … Si nos fijamos, desde el vértice v_0 podemos alcanzar todos los vértices.

f) Caminos mínimos (con pesos) desde v_0 al resto de vértices:

$\langle v_0, v_1\rangle$, $\langle v_0, v_3, v_2\rangle$, $\langle v_0, v_3\rangle$, $\langle v_0, v_3, v_4\rangle$, $\langle v_0, v_3, v_6, v_5\rangle$, $\langle v_0, v_3, v_6\rangle$

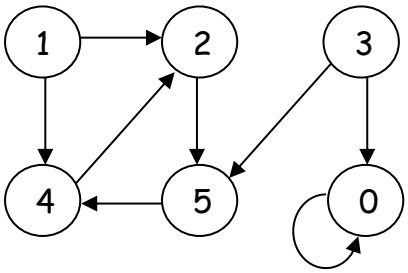
g) ¿Tiene ciclos?

Sí. Por ejemplo: $\langle v_0, v_3, v_2, v_0\rangle$

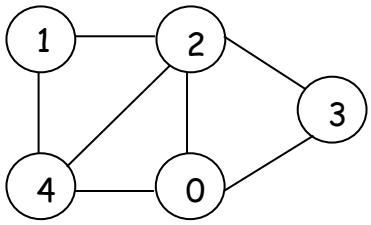
EJERCICIO 2

Representad los siguientes grafos mediante una matriz de adyacencia y mediante listas de adyacencia:

a)



b)

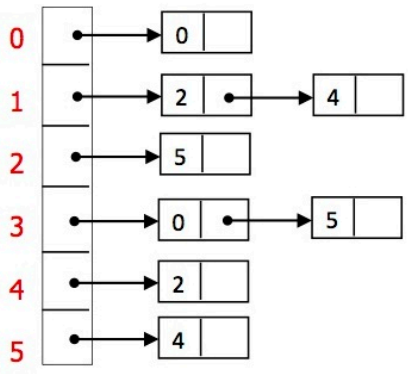


Solución

Recordad que si se representa el grafo mediante una matriz de adyacencia, los índices de las filas son los vértices origen y los índices de las columnas son los vértices destino. El valor de la matriz es true o false (si no hay pesos) o si hubiera peso entonces se pone el valor del peso. Fijaos también que en el grafo no dirigido si hay una arista (u,v) también existe la arista (v,u).

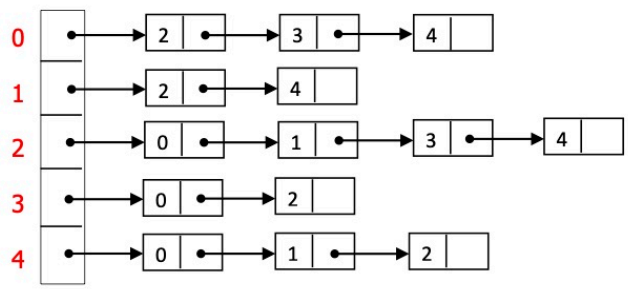
a)

	0	1	2	3	4	5
0	true	false	false	false	false	false
1	false	false	true	false	true	false
2	false	false	false	false	false	true
3	true	false	false	false	false	true
4	false	false	true	false	false	false
5	false	false	false	false	true	false



b)

	0	1	2	3	4
0	false	false	true	true	true
1	false	false	true	false	true
2	true	true	false	true	true
3	true	false	true	false	false
4	true	true	true	false	false



EJERCICIO 3

Diseña los siguientes métodos en la clase *GrafoDirigido*:

a) Consultar el grado de salida de un vértice dado.
b) Consultar el grado de entrada de un vértice dado.
c) Empleando los dos métodos anteriores, escribir un método que devuelva el grado del grafo.
d) Comprobar si un vértice es *fuente*, es decir, si es un vértice del que sólo salen aristas.
e) Comprobar si un vértice es un *sumidero* (i.e. un vértice al que sólo llegan aristas) al que llegan aristas de todos los demás vértices del grafo.

Solución

a) Consultar el grado de salida de un vértice dado. El grado de salida de un vértice es el número de arcos, o sea de adyacentes, que tiene. Como está representado como un array de listas, entonces el grado de salida es la longitud de la lista de adyacencia correspondiente a ese vértice.

```
public int gradoDeSalida(int v) {
  return elArray[v].talla();
}
```

b) Consultar el grado de entrada de un vértice dado. Obtener el grado de entrada ya no es tan inmediato porque con la representación del grafo con listas de adyacencia sólo tenemos los vértices adyacentes a uno dado (las aristas que salen, no las aristas que entran). Por eso hay que recorrer todos los vértices del grafo y para cada vértice recorrer todos los arcos que salen de él, y acumular en una variable todos los arcos que llevan al vértice v dado. El coste de esta operación será O(|V| + |A|). Es una operación costosa. ¿Se os ocurre alguna forma de mejorar este coste si es que esta operación es frecuente en nuestra aplicación? Al igual que tenemos una implementación del grafo con listas de adyacentes, podríamos tener una representación con listas de incidentes y el método sería análogo al caso anterior: devolver la talla de la lista de incidencia del vértice.

```
public int gradoDeEntrada(int v) {
      int res = 0;
      for (int i = 0; i < numV; i++) {
        ListaConPI<Adyacente> ady = elArray[i];
        for (ady.inicio(); !ady.esFin(); ady.siguiente())
          if (ady.recuperar().destino == v) {
            res++;
            ady.fin();
          }
      }
      return res;
    }
```

c) Empleando los dos métodos anteriores, escribir un método que devuelva el grado del grafo. El grado de un vértice es la suma del grado de entrada y del grado de salida, y el grado de un grafo es el grado del vértice de mayor grado.

```
public int gradoGrafo() {
  int res = 0;
  for (int v = 0; v < numV; v++) {
    int gradoV = gradoDeEntrada(v) + gradoDeSalida(v);
    if (gradoV > res) res = gradoV;
  }
  return res;
}
```

***Nota*:** sin invocar a los dos métodos anteriores es posible implementar este método mucho más eficientemente.

d) Comprobar si un vértice es *fuente*, es decir, si es un vértice del que sólo salen aristas. Para saber si un vértice es fuente habrá que recorrer todo el grafo: todas las listas de vértices adyacentes y comprobar que el vértice dado no aparece en ninguna lista, es decir, que no existe ninguna arista que tenga como destino el vértice indicado. De nuevo, el coste de esta operación será el coste recorrer todo el grafo, O(|V| + |A|).

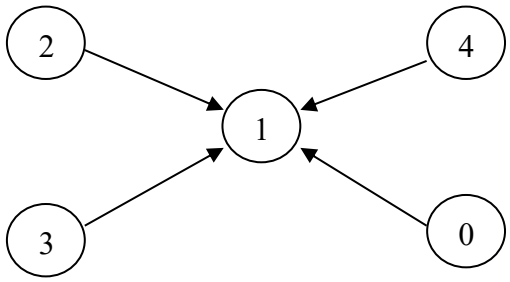
Fijaos que también podríamos reutilizar el método que calcula el grado de entrada de un vértice para comprobar si el vértice es fuente: basta con comprobar que el grado de entrada sea 0. Pero la solución que planteamos a continuación es un poco más eficiente: si en un momento dado del recorrido del grafo se encuentra una arista incidente al vértice (esto es, ese vértice aparece como adyacente de otro en el grafo), el método poner la variable booleana "res" a false y finalzai, sin necesidad de recorrer el grafo en su totalidad.

```
public boolean esFuente(int v) {
  boolean res = !elArray[v].esVacia();
  for (int i = 0; i < numV && res; i++) {
    ListaConPI<Adyacente> ady = elArray[i];
    for (ady.inicio(); !ady.esFin(); ady.siguiente())
      if (ady.recuperar().destino == v) {
        res = false;
        ady.fin();
      }
  }
  return res;
}
```

e) Comprobar si un vértice es un *sumidero* (i.e. un vértice al que sólo llegan aristas) al que llegan aristas de todos los demás vértices del grafo. Para ver si un vértice v es sumidero primero se comprueba que no sale de él ninguna arista (elArray[v] es una lista vacía). Luego se recorren todos los vértices (menos él mismo) y se comprueba que en todos ellos hay un arco que lleva a v.

De nuevo, como en el caso anterior, se podrían utilizar los métodos que calculan el grado de entrada y salida: después de comprobar que no hay aristas que salen de él (res = elArray[v].esVacia();), se podría saber si un vértices es sumidero comprobando que el número de aristas que inciden en él es el número de vértices menos 1, es decir, que su grado de entrada es |V|-1. De nuevo, es más eficiente la solución planteada porque si no se encuentra una arista en uno de los vértices no es necesario seguir explorando el grafo.

Ejemplo: el vértice 1 es un sumidero de este tipo.




```
public boolean sumideroCompleto(int v) {
  boolean res = elArray[v].esVacia();
  for (int i = 0; i < numV && res; i++) {
    if (i != v) {
      boolean llegaArista = false;
      ListaConPI<Adyacente> ady = elArray[i];
      for (ady.inicio(); !ady.esFin(); ady.siguiente())
        if (ady.recuperar().destino == v) {
          llegaArista = true;
          ady.fin();
        }
      if (!llegaArista) res = false;
    }
  }
  return res;
}
```

EJERCICIO 4

Un grafo transpuesto T de un grafo G tiene el mismo conjunto de vértices pero con las direcciones de las aristas en sentido contrario, es decir, que una arista (u, v) en G se corresponde con una arista (v, u) en T.

Diseña un método en la clase *GrafoDirigido* que permita obtener su grafo transpuesto:

```
public GrafoDirigido grafoTranspuesto();
```

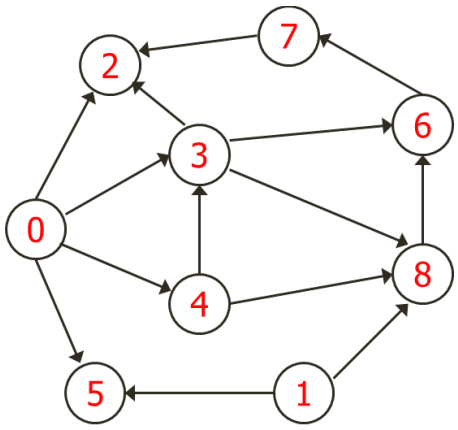
Solución

Lo que habrá que hacer es construir un grafo con el mismo número de vértices, sin aristas. Después, se recorre el grafo original: cada vez que encontremos una arista (i,j) en el grafo original, insertar la (j,i) en el grafo traspuesto.

```
public GrafoDirigido grafoTranspuesto() {
  GrafoDirigido res = new GrafoDirigido(numV);
  for (int i = 0; i < numV; i++) {
    ListaConPI<Adyacente> ady = elArray[i];
    for (ady.inicio(); !ady.esFin(); ady.siguiente()) {
      Adyacente a = ady.recuperar();
      res.insertarArista(a.destino, i, a.peso);
    }
  }
  return res;
}
```

EJERCICIO 5

Mostrar el resultado de imprimir por pantalla el recorrido en profundidad y en anchura del siguiente grafo:



Solución

Nota: el orden de las aristas afecta al resultado de los recorridos. La solución que se muestra aquí asume que en la listas de adyacencia las aristas están ordenadas por el código del vértice destino de menor a mayor.

Profundidad (DFS): 0-2-3-6-7-8-4-5-1

Amplitud (BFS): 0-2-3-4-5-6-8-7-1

Para el recorrido en profundidad (DFS):

- Empieza por el 0 y se lanza un bucle para recorrer todos sus adyacente (2,3,4). Lista {0}
- Cuando se empieza por el primer adyacente (el 2), al ser un recorrido en profundidad, de nuevo se lanza un bucle para recorrer todos sus adyacentes (ninguno en este caso). Tened en cuenta que están pendientes los vértices (3 y 4) de la etapa anterior. Lista {0,2}.
- Como el 2 no tenía adyacentes se sigue por el 3 (que es el primero que estaba pendiente de los adyacentes del 0. De nuevo se lanza un bucle con los adyacentes del 3 (2,6,8), y se empieza por el 2 Lista {0,2,3}.
- PERO el 2 ya había sido visitado, por lo que se ignora. Y se sigue por el 6, y se inicia un bucle con los adyacentes del 6 (el 7). Lista {0,2,,3,6}
- etcétera…

Fijaos en una cosa importante. Después de este recorrido tenéis en la lista {0,2,3,6,7,8,4,5} PERO falta el 1, porque no ha sido alcanzable desde el 0. Por eso el algoritmo de recorrido tiene una lanzadera, que es un bucle que intenta hacer el recorrido desde todos los nodos que aún no han sido visitados. Se había lanzado desde el vértice 0 y ahora se lanza desde el 1. Se añade el 1 y se sigue lanzando desde los demás, pero como ya han sido visitados no se hace nada.

Para el recorrido en anchura (BFS):

- Aquí, empezamos también por el 0. Lo metemos en la cola. Ahora el proceso es dsencolar, apuntar el nodo desencolarlo y meter a sus adyacentes (2,3,4) en una cola
- Lista {0}
- Ahora se saca el primero de la cola, el 2, se mete en la lista {0,2} y se meten sus adyacentes (ninguno) en la cola, de modo que la cola queda con (3,4).
- Y así sucesivamente.

Fijaos que de nuevo el 1 no entra, y se añade al final porque se vuelve a lanzar el recorrido desde el 1.

EJERCICIO 6

Implementa un método en la clase *Grafo* que compruebe si un vértice es alcanzable desde otro vértice dado.

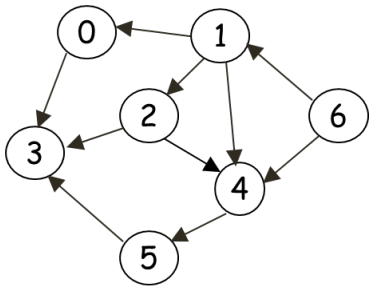
Solución

Este algoritmo no es más que una adaptación del recorrido en profundidad DFS. La idea es partir de vOrigen y detectar si durante ese recorrido se alcanza vDestino.
Se lanza el algoritmo recursivo, desde vOrigen y pasando como otro parámetro vDestino. La idea del algoritmo DFS es partir de un vértice recorrer todos sus adyacentes y seguir recursivamente la búsqueda desde cada adyacente. En cada llamada recursiva el vOrigen es el nuevo vértice alcanzado. Por eso se pregunta si vOrigen== vDestino, en cuyo caso ya hemos encontrado que es alcanzable.
Todas las demás líneas corresponden al recorrido en profundidad. Fijaos que para cada adyacente del vértice que estamos analizando en una determinada llamada recursiva hay que comprobar que no ha sido ya visitado (para no entrar en bucle).

```
public boolean esAlcanzable(int vOrigen, int vDestino) {
  visitados = new int[numVertices()];
  return esAlcanzableRec(vOrigen, vDestino);
}

private boolean esAlcanzableRec(int vActual, int vDestino) {
  if (vActual == vDestino) return true;
  visitados[vActual] = 1;
  ListaConPI<Adyacente> ady = adyacentesDe(vActual);
  for (ady.inicio(); !ady.esFin(); ady.siguiente()) {
    int vSiguiente = ady.recuperar().destino;
    if (visitados[vSiguiente] == 0 && esAlcanzableRec(vSiguiente, vDestino))
      return true;
  }
  return false;
}
```

EJERCICIO 7

Siguiendo el método *ordenacionTopologica*, mostrar la ordenación topológica resultante para el siguiente grafo dirigido acíclico. ¿La ordenación obtenida es única? En caso negativo mostrar otra ordenación válida.

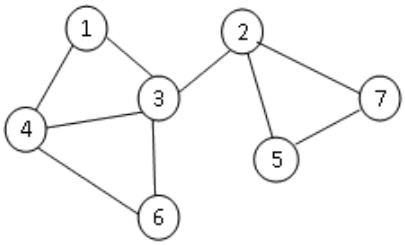


Solución

Ordenación obtenida: $6 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 0 \rightarrow 3$
No es la única ordenación posible. Ejemplo: $6 \rightarrow 1 \rightarrow 0 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3$

EJERCICIO 8

En teoría de grafos un puente es una arista tal que si fuese eliminada de un grafo incrementaría el número de componentes conexas de éste. Nótese entonces que un grafo conexo dejaría de serlo si se elimina una arista puente; por ejemplo, la arista (2, 3) del grafo de la siguiente figura es un puente.



Se pide diseñar un método en la clase Grafo que compruebe si una arista (i, j) es una arista puente.

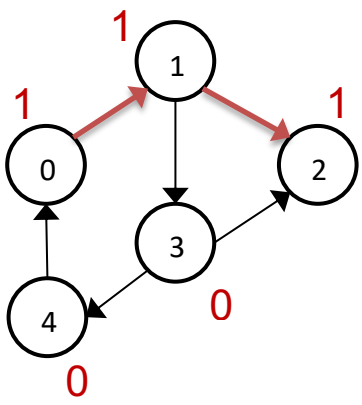
Solución

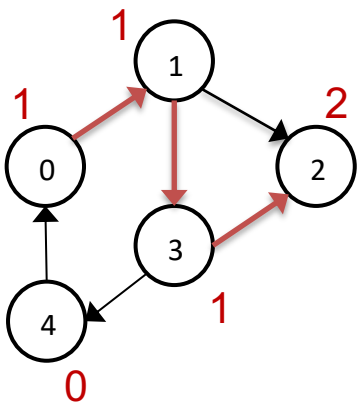
```
public boolean esAristaPuente(int i, int j) {
  visitados = new int[numVertices()];
  return esAristaPuente(i, i, j);
}

protected void esAristaPuente(int v, int i, int j) {
  visitados[v] = 1;
  ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (v != i || w != j) { // No es la arista (i,j)
      if (w == j) return false; // Otro camino lleva a j
      if (visitados[w] == 0 && !esAristaPuente(w, i, j))
        return false;
    }
  }
  return true;
}
```

EJERCICIO 9

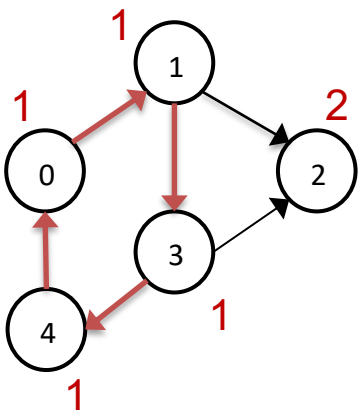
Diseña un método en la clase Grafo que compruebe si el grafo tiene ciclos.

Para ello no basta con saber si un vértice ha sido visitado o no, necesitamos tres posibles estados: **0** (no visitado), **1** (visitado en el camino actual), **2** (visitado por otro camino).





No es un ciclo, el vértice 2 fue visitado por otro camino



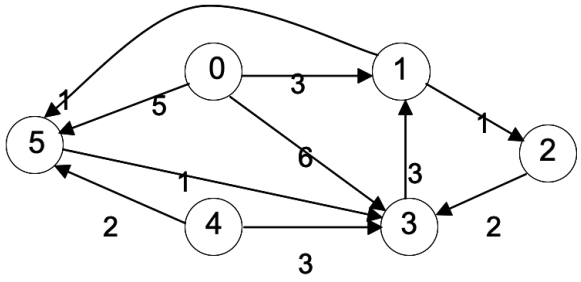
Ciclo detectado: una arista vuelve a un vértice visitado en el camino actual

Solución

```
public boolean tieneCiclos() {
  boolean ciclo = false;
  visitados = new int[numVertices()];
  for (int v = 0; v < numVertices() && !ciclo; v++)
    if (visitados[v] == 0) ciclo = tieneCiclos(v);
  return ciclo;
}

protected boolean tieneCiclos(int v) {
  boolean ciclo = false;
  visitados[v] = 1; // En el camino actual
  ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin() && !ciclo; l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 0) ciclo = tieneCiclos(w);
    else if (visitados[w] == 1) ciclo = true;
  }
  visitados[v] = 2; // Visitado por otro camino
  return ciclo;
}
```

EJERCICIO 10

Los vértices del siguiente grafo representan personas (Ana, Begoña, Carmen, Daniel, Eliseo y Francisco) y las aristas indican si una persona tiene el número de móvil de otra. El peso de una arista es el coste de enviar un SMS (por ejemplo, Ana puede enviar un SMS a Begoña por 3 céntimos). Haz una traza de Dijkstra para averiguar cuál sería la forma más barata de que Ana le haga llegar un SMS a Francisco. (Sea Ana el vértice 0, Begoña 1, Carmen 2, Daniel 3, Eliseo 4 y Franncisco 5, respectivamente.)



Solución

Veremos algunos pasos del algoritmo. Recuerdo que:

- V: es el vértice de mínimo valor en la cola de prioridad (es decir, el que tiene el camino más corto desde el origen) y que ya puedo fijar su coste como DEFINITIVO, no es mejorable.
- qPrior: Cola de prioridad (minHeap) con los costes obtenidos hasta el momento para llegar a los vértices.
- Vector visitados: Marca lo vértices que ya están fijos sus valores, y no pueden mejorar.
- Vector distanciaMin: Las distancias obtenidas hasta el momento para llegar a todos los vértices.
- Vector caminoMin: Se apunta desde qué vértice anterior se ha obtenido ese valor minimo que está almacenado en distanciaMin.

Empezamos ahora la traza:

- Inicialización: se inserta (0,0) en el minHeap.
- Primera iteración del bucle: Se extrae (0,0) del minHeap, actualiza el vértice 0 a visitado. Explora los adyacentes a 0: 1, 3 y 5, actualizando sus distancias, sus predecesores e insertando las tuplas en el minHeap. Como en los tres casos se viene desde 0 se pone 0 en la correspondiente casilla de caminoMin.
- En la siguiente línea se extrae el mínimo de la cola de prioridad, que es el vértice 1 con peso 3. Se marca como fijo el 1 (verdadero en visitados). Se calcula para cada uno de sus adyacentes (2 y 5) la mejor forma de llegar hasta ellos (el mínimo de lo que ya había o el nuevo camino obtenido 0-1-2 0-1-5. Se modifica también caminoMin.
- Así sucesivamente…

Asumimos que los códigos de los vértices son: a = 0, b = 1, c = 2, d = 3, e = 4, f = 5.

V	qPrior	visitados						distanciaMin						caminoMin					
		0	*1*	*2*	*3*	*4*	*5*	*0*	*1*	*2*	*3*	*4*	*5*	*0*	*1*	*2*	*3*	*4*	*5*
	(0,0)	0	0	0	0	0	0	0	∞	∞	∞	∞	∞	-1	-1	-1	-1	-1	-1
0	**(1,3)**, (3,6), (5,5)	1	0	0	0	0	0	0	3	∞	6	∞	5	-1	0	-1	0	-1	0
1	(3,6), (5,5), **(2,4)**, (5,4)	1	1	0	0	0	0	0	3	4	6	∞	4	-1	0	1	0	-1	1
2	(3,6), (5,5), **(5,4)**	1	1	1	0	0	0	0	3	4	6	∞	4	-1	0	1	0	-1	1
5	(3,6), (5,5), **(3,5)**	1	1	1	0	0	1	0	3	4	5	∞	4	-1	0	1	5	-1	1
3	(3,6), **(5,5)**	1	1	1	1	0	1	0	3	4	5	∞	4	-1	0	1	5	-1	1
5	**(3,6)**	Vértice v_5 ya visitado																	
3	∅	Vértice v_3 ya visitado																	

La forma más barata de que Ana (vértice origen, 0) le haga llegar un SMS a Francisco (vértice 5) es: Ana (vértice 0) → Begoña (vértice 1) → Francisco (vértice 5), con un coste 4.