EI1013/MT1013 ESTRUCTURA DE DATOS

BOLETÍN DE PROBLEMAS

GRAFOS

En este boletín se recopilan ejercicios de examen. Todos los ejercicios han aparecido en algún examen final de la asignatura. éste que se indica en la línea final del enunciado. Además, en cada ejercicio se indica el peso que tuvo ese ejercicio sobre la nota máxima del examen, siempre 10 puntos. Para una parte de los ejercicios se incluyen posibles soluciones.

4. 1 **(3 puntos)** Deseamos almacenar los prerrequisitos para matricularse en nuevas asignaturas de un grado. Una asignatura es prerrequisito de otra si es necesario haberla superado para poder matricularse de la segunda. Para almacenar esta información decidimos usar un grafo dirigido no ponderado, implementado mediante listas de adyancencia, **ListGraph<String, Object>**, en el que los códigos de las asignaturas se representarán mediante **String**. Este grafo almancenará todos los prerrequisitos del plan de estudios de un grado. Los arcos representarán las dependencias entre asignaturas. Por ejemplo, si hay un arco de `A` a `B` significa que para poder matricularse de la asignatura de código `B` se tiene que tener aprobada la asignatura `A`.

La parte privada de la clase **ListGraph** es la vista en clase:

```
public class ListGraph<T, W> implements EDGraph<T, W>{
    private class Node<T> {
       public T data;
       public List<EDEdge<W>> lEdges;
       public Node (T data) {
          this.data = data;
          this.lEdges = new LinkedList<EDEdge<W>>();
       }
    }

    private ArrayList<Node<T>> nodes;
}

public class EDEdge<W> {
    public int source, target;
    public W weigth;
}
```

(a) **(1,5 puntos)** Añadir una operación a la clase **ListGraph** tal que dada la etiqueta de un nodo del grafo, devuelva el conjunto de etiquetas de sus nodos predecesores inmediatos. Es decir, en el caso de antes, **predecessors ('B')** devolverá un conjunto con `'A'` y las etiquetas de todos los nodos que tengan arcos a `'B'`. Si el grafo no contiene ningún nodo con la etiqueta que se pasa como parámetro, devolverá un conjunto vacío.

```
public Set<T> predecessors(T item);
```

(b) **(1,5 puntos)** Implementa el método estático **matriculaPosible** que tome como parámetros un grafo de prerrequisitos, una lista con las asignaturas de las que se desea matricular, y las asignaturas ya aprobadas. Devolverá un booleano indicando si puede o no matricularse de todas las asignaturas deseadas. Para almacenar las asignaturas aprobadas elige la colección más eficiente para implementar el algoritmo.

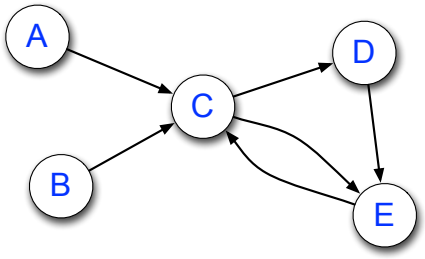
Curso 2016/17 Examen Final 1ª Convocatoria

Solución:

```
public Set<T> predecesores (T item) {
    HashSet<T> s = new HashSet<T>();
    int p= this.getNodeIndex(item);
    if (p==-1) throw new NoSuchElementException();
    for (int i=0; i<nodes.size(); i++) {
        if (i!=p && nodes.get(i).data!=null) {
            Iterator<EDEdge<W>> it = nodes.get(i).lEdges.listIterator();
            boolean stop=false;
            while (it.hasNext() && !stop) {
                EDEdge<W> e=it.next();
                if (e.getTarget()==p) {
                    s.add(nodes.get(i).data);
                    stop=true; //solo hay un arco entre cada par de nodos
                }
            }
        }
    }
    return s;
}

public static boolean matriculaPosible (EDGraph<String, Object> requisitos,
                    List<String> matricula; Set<String> aprobadas) {
    for (String asig: matricula) {
        Set<String> prev = g.predecessors(asig);
        if (!aprobadas.containsAll(prev)
            return false;
    }
    return true;
}
```

4. 2 **(2.5 puntos)** Dada la clase que implementa un grafo dirigido mediante listas de adyacencia, **ListGraph**, implementa un método en dicha clase que devuelva una estructura con los índices de los nodos cuyo grado de entrada sea 0. Por ejemplo, para el grafo de la figura, el método debería devolver los índices de los nodos etiquetados con las letras 'A' y 'B'.



Considerar que la parte privada de la clase **ListGraph** es:

```
public class ListGraph<T,W> implements EDGraph<T,W>{
    private class Node<T> {
       T data;
       List<EDEdge<W>> lEdges;
       Node (T data) {
          this.data = data;
          this.lEdges = new LinkedList<EDEdge<W>>();
       }
    }

    private int size;
    private ArrayList<Node<T>> nodes;
}

public class EDEdge<W> {
    public int source, target;
    public W weight;
}
```

Curso 2015/16 Examen Final 1ª Convocatoria

Solución:

```
public Set<Integer> getOriginNodes() {
    Set<Integer> s=new HashSet<Integer>();
    for (int i=0; i<this.nodes.size(); i++)
        if (nodes.get(i).data!=null)
            s.add(i);
    int i=0;
    while (!s.isEmpty() && i<this.nodes.size()) {
        if (this.nodes.get(i).data!=null) {
            Iterator<EDEdge<W>> it= this.nodes.get(i).lEdges.listIterator();
            while (it.hasNext() && !s.isEmpty())
                s.remove(it.next().getTarget());
        }
        i++;
    }

    return s;
}
```

4. 3 **(2 puntos)** Dada un implementación de grafos mediante listas de adyancencia:

```
public class EDListGraph<T,W> implements EDGraph<T,W> {
    private static int DEFAULT_SIZE = 20;

    private class Node<T> {
        T data;
        List< EDEdge<W> > lEdges;

        Node (T data);
    }

    private int size; //real number of nodes
    private Node<T>[] nodes;
```

Implemeta un método tal que dado un nodo del grafo, devuelva **true** si dicho nodo participa de un ciclo y **false** en caso contrario. Para ello tendrás que adaptar el método de búsqueda en profundidad

```
public boolean takesPartInACycle (int vertex);
```

Curso 2014/15 Examen Final 2ª Convocatoria

Solución:

Versión recursiva

```
public boolean takesPartInACycle (int vertex) {
        if (vertex<0 || vertex >= nodes.length;
                return false;

        boolean []  visited = new boolena[nodes.length];
        for (int i=0; i<visited.length; i++)
                visited[i]=false;
        int current = vertex;
        return takesPartInACycle(vertex, current, visited);
}

private boolean takesPartInACycle (int vertex, int current, boolean [] visited) {
        boolean cycle = false;

        visited[current] = true;
        for (EDEdge<W> edge: nodes[current].lEdges)
                if (edge.getTarget()==vertex)
                        return true;
                else (if (!visited[edge.getTarget()])
                        cycle = takesPartInACycle(vertex, edge.getTarget(), parent);
                        if (cycle) break;
                }
        return cycle;
}
```

Versión no recursiva.

```
public boolean takasPartInACycle(int vertex) {
        if (vertex<0 || vertex >= nodes.length;
                return false;

        Stack<Integer> s = new Stack<Integer>;
        boolean visited[] = new boolean[nodes.length]

        s.push(vertex);
        while(!s.empty()) {
                int current = s.pop();
                visited[current] = true;
                for (EDEdge<W> edge: nodes[current].lEdges) {
                        int target = edge.getTarget()
                        if (target == vertex)
                                return true;

                        if (!visited[target]) {
                                visited[target] = true;
                                s.push(target);
                        }
                }
        }

        return false;

}
```

4. 4 **(2 puntos)** Supongamos que para representar los grafos como listas de adyacencia, utilizamos la siguiente clase con los atributos privados que se indican:

```
public class Edge<T, W> {
    public T target;  //destination vertex of the edge
    public T source;   //source vertex of the edge

    public Edge(T source, T target); //Constructor

    public boolean equals (Object other);
}

public class ListGraph<T,W> {
    // Private data
    private HashMap<T, List<Edge<T,W>>> nodes;
    private boolean directed;
```

(a) **(1 punto)** Implementar el método de la clase **insertNode**, con el siguiente interfaz:

```
public boolean insertNode(T item);
```

El método devolverá **true** si se ha insertado y **false** en caso contrario.

(b) **(1 punto)** Implementar el método que inserta un arco en un grafo, tanto si es dirigido como si no. Devolverá **true** si se ha insertado y **false** en caso contrario.

```
public boolean insertEdge(T fromNode, T toNode);
```

Curso 2014/15 Examen Final 1ª Convocatoria

Solución:

```
public boolean insertNode(T item) {
    if (!nodes.containsKey(item)) {
        nodes.put(item, new LinkedList<Edge<T,W>>());
        return true;
    }
    else return false;
}

public boolean insertEdge(T fromNode , T toNode ) {
    if (nodes.containsKey(fromNode) && nodes.containsKey(toNode)) {
        List<Edge<T,W>> ledges = nodes.get(fromNode);
        Edge<T,W> edge = new Edge<T,W>(fromNode, toNode);
        if (!ledges.contains(edge)) {
            ledges.add(edge);
            nodes.put(fromNode, ledges);
            if (!directed) {
                Edge<T,W> reverse = new Edge<T,W>(toNode,fromNode);
                nodes.get(toNode).add(reverse);
            }
            return true;
        }
    }
    return false;
}
```

4. 5 **(3 puntos)** Deseamos almacenar los prerrequisitos para matricularse en nuevas asignaturas de un grado. Una asignatura es prerrequisito de otra si es necesario haberla superado para poder matricularse de la segunda. Para almacenar esta información decidimos usar un grafo dirigido no ponderado, implementado mediante listas de adyancencia, **ListGraph<String, Object>**, en el que los códigos de las asignaturas se representarán mediante **String**. Este grafo almancenará todos los prerrequisitos del plan de estudios de un grado. Los arcos representarán las dependencias entre asignaturas. Por ejemplo, si hay un arco de `A` a `B` significa que para poder matricularse de la asignatura de código `B` se tiene que tener aprobada la asignatura `A`.

La parte privada de la clase **ListGraph** es la vista en clase:

```
public class ListGraph<T, W> implements EDGraph<T, W>{
    private class Node<T> {
       public T data;
       public List<EDEdge<W>> lEdges;
       public Node (T data) {
          this.data = data;
          this.lEdges = new LinkedList<EDEdge<W>>();
       }
    }

    private ArrayList<Node<T>> nodes;
}

public class EDEdge<W> {
    public int source, target;
    public W weigth;
}
```

(a) **(1,5 puntos)** Añadir una operación a la clase **ListGraph** tal que dada la etiqueta de un nodo del grafo, devuelva el conjunto de etiquetas de sus nodos predecesores inmediatos. Es decir, en el caso de antes, **predecessors ('B')** devolverá un conjunto con `'A'` y las etiquetas de todos los nodos que tengan arcos a `'B'`. Si el grafo no contiene ningún nodo con la etiqueta que se pasa como parámetro, devolverá un conjunto vacío.

```
public Set<T> predecessors(T item);
```

(b) **(1,5 puntos)** Implementa el método estático **matriculaPosible** que tome como parámetros un grafo de prerrequisitos, una lista con las asignaturas de las que se desea matricular, y las asignaturas ya aprobadas. Devolverá un booleano indicando si puede o no matricularse de todas las asignaturas deseadas. Para almacenar las asignaturas aprobadas elige la colección más eficiente para implementar el algoritmo.

Curso 2016/17 Examen Final 1ª Convocatoria

Solución:

```
public Set<T> predecesores (T item) {
    HashSet<T> s = new HashSet<T>();
    int p= this.getNodeIndex(item);
    if (p==-1) throw new NoSuchElementException();
    for (int i=0; i<nodes.size(); i++) {
        if (i!=p && nodes.get(i).data!=null) {
            Iterator<EDEdge<W>> it = nodes.get(i).lEdges.listIterator();
            boolean stop=false;
            while (it.hasNext() && !stop) {
                EDEdge<W> e=it.next();
                if (e.getTarget()==p) {
                    s.add(nodes.get(i).data);
                    stop=true; //solo hay un arco entre cada par de nodos
                }
            }
        }
    }
    return s;
}

public static boolean matriculaPosible (EDGraph<String, Object> requisitos,
                    List<String> matricula; Set<String> aprobadas) {
    for (String asig: matricula) {
        Set<String> prev = g.predecessors(asig);
        if (!aprobadas.containsAll(prev)
            return false;
    }
    return true;
}
```

4. 6 **(2 puntos)** En un grafo implementado con listas de adyacencia, añadea la clase **EDListGraph<T,W>** un método que devuelva una lista con los índices de los nodos que **NO** son alcanzables desde el nodo **start** .

```
public List<Integer> notReachable (int start)
```

La parte privada de la clase **EDListGraph<T,W>** es la siguiente:

```
public class EDListGraph<T,W> implements EDGraph<T,W> {
    private class Node<T> {
        T data;
        List< EDEdge<W> > lEdges;

        Node (T data) {
            this.data = data;
            this.lEdges = new LinkedList< EDEdge<W> >();
        }
        public boolean equals (Object other) {
            //equals for node
            ...
        }
    }
    private ArrayList<Node<T>> nodes;  //Vector of nodes
    private int size; //number of nodes
    private boolean directed;
```

El vector de nodos, **nodes**, está inicializado a **null** y cuando un nodo es borrado también se pone a **null** la posición correspondiente. **Nota:** En la implementación de este método solo puedes usar los métodos de la interfaz**EDGraph** que aparecen en la nota de interfaces. Cualquier otro método que se use hay que implementarlo.

Curso 2017/18 Examen Final 2ª Convocatoria

4. 7 **(3 puntos)** Usamos un grafo para representar una red social igual que se ha hecho en las prácticas. Cada persona corresponde a un nodo y los arcos representan la relación de amistad. Añade un método público a la clase **EDListGraph** que a partir de la posición del nodo correspondiente a una determinada persona, **start** , devuelva el conjunto de personas con un mayor grado de separación (distancia que los separa, como en la práctica) respecto a la primera. Si **start** contiene un valor no válido el resultado será **null** .

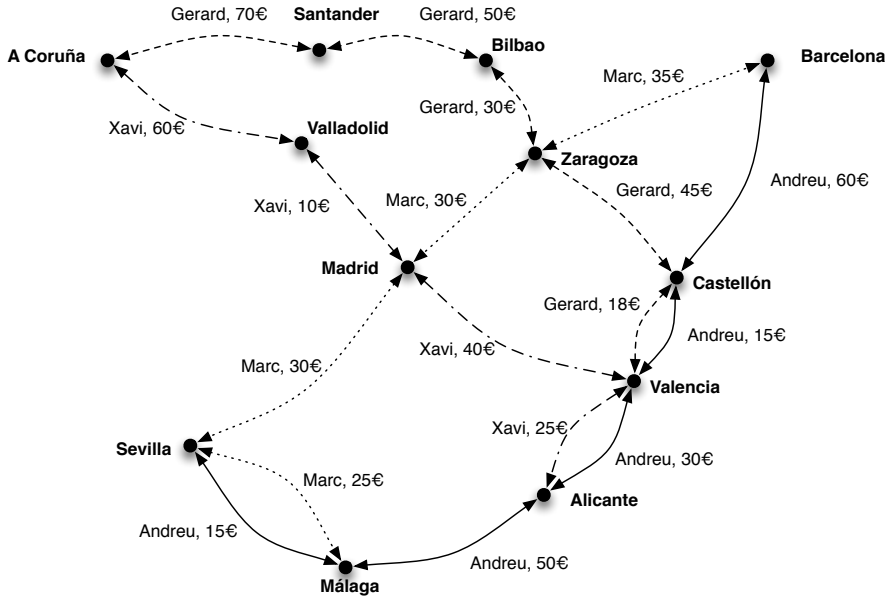
```
public Set<T> greaterDegreeSep (int start);
```

Nota: debes implementar cualquier método que no aparezca en la interfaz **EDGraph** que se proporciona. Los atributos de la clase **EDListGraph** son:

```
public class EDListGraph<T,W> implements EDGraph<T,W> {
        private class Node<T> {
                T data;
                List< EDEdge<W> > lEdges;
                Node (T data) {...}
                public boolean equals (Object other) {...}
        }
        private ArrayList<Node<T>> nodes;
        private int size;
}
```

Curso 2017/16 Examen Final 1ª Convocatoria

4. 8 Una empresa de transportes dispone de varias rutas entre ciudades servidas por varios camioneros autónomos. Cada chofer tiene una tarifa entre cada par de ciudades. Entre un par de ciudades concretas pueden existir varios camioneros que la incluyan en su ruta. Para almacenar esta información decidimos usar un grafo en el que cada nodo represente una ciudad y cada arco almacene, el nombre del camionero y la tarifa entre ese par de ciudades.



(a) **(1 punto)** Dado el ejemplo anterior realiza un recorrido en anchura empezando en *Castellón* y teniendo en cuenta que cada nodo sólo se visita una vez. Indica claramente la secuencia de nodos visitados en cada paso. Muestra también el árbol de recubrimiento resultante.

(b) **(1.5 puntos)** Para implementar el grafo definimos la clase **Tramo** que representa los datos del peso de un arco, y definimos el grafo usando la interfaz **EDGraph** y **EDEdge**, que puedes encontrar en la hoja adjunta de interfaces.

```
public class Tramo {
                public String chofer;
                public int tarifa;
        }
```

```
Un grafo de rutas se define como:

    EDGraph<String,Tramo>
```

Escribe un método estático que, tomando un grafo y el nombre de un ciudad, escriba por pantalla el tramo de menor coste que salga de ella. Deberá imprimir la ciudad a la que se llega, el chofer de esa ruta y su coste.

Curso 2013/14 Examen Final 1ª Convocatoria

4. 9 **(2,5 puntos)** Vamos a utilizar un grafo para representar una red social. Cada persona corresponde a un nodo y los arcos representan la relación de amistad. Añade un método público a la clase **EDListGraph** que a partir de la posición del nodo correspondiente a una determinada persona **start** , permita encontrar y listar por pantalla los nombres de los amigos que estén a una distancia menor o igual que **k** (siendo **k** un entero positivo mayor que 0).

```
public void kDistance(int start, int k);
```

Por ejemplo, si A es amigo de B, su distancia es 1 y los amigos de B, que no sean amigos de A, están a una distancia 2 de A.

```
public class EDListGraph<T,W> implements EDGraph<T,W> {
    private class Node<T> {
        T data;
        List<EDEdge<W> > lEdges;
    }

    // Private data
    private ArrayList<Node<T>> nodes;
    private int size; //real number of nodes
    private boolean directed;
    ...
}
```

Curso 2016/17 Examen Final 2ª Convocatoria

4. 10 **(2 puntos)** Dada la clase **ListGraph** que implementa un grafo dirigido mediante listas de adyacencia, implementad un método **entryDegree** para dicha clase tal que tome el índice de un nodo devuelva el grado de entrada de éste.

La parte privada de la clase **ListGraph** es:

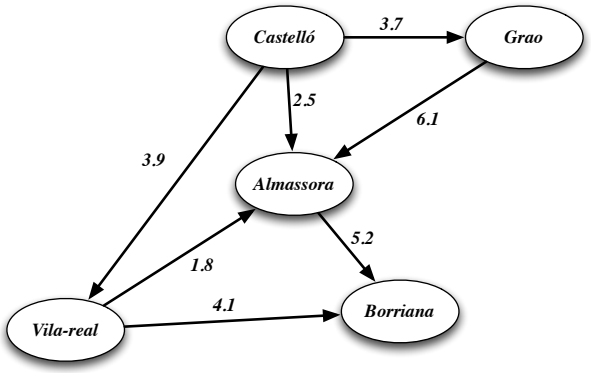
```
public class ListGraph<T, W> {
    private class Node<T> {
       T data;
       List<EDEdge<W>> lEdges;
       Node (T data) {
          this.data = data;
          this.lEdges = new LinkedList<EDEdge<W>>();
           }
    }

    private int size;
    private ArrayList<Node<T>> nodes;
}

public class EDEdge<W> {
    public int source, target;
    public W weight;
}
```

Curso 2015/16 Examen Final 2ª Convocatoria

4. 11 **(2,5 puntos)** En un fichero de texto se tiene toda la información de un grafo dirigido y ponderado, donde los pesos serán valores enteros. En la primera línea del fichero estará el número de nodos del grafo. A continuación, la etiqueta **String** de cada nodo, cada dato en una línea distinta. Finalmente se encuentra la información de las listas de adyacencia de cada nodo. Cada lista de adyacencia está en una línea y hay una línea por cada nodo (aunque no tenga arcos). En cada línea con la lista de adyacencia, en primer lugar, estará el número de arcos y después los arcos. Cada arco consta de etiqueta del nodo destino y peso del arco. Los datos de los arcos en una lista están separados por uno o más espacios en blanco. Si no tiene lista de arcos, la línea correspondiente tendrá sólo el número 0 (0 arcos). Por ejemplo, el grafo de la izquierda se describiría mediante el fichero de la derecha:




```
5
Castelló
Grao
Almassora
Borriana
Vila-real
3 Grao 3.7 Almassora 2.5 Vila-real 3.9
1 Almassora 6.1
1 Borriana 5.2
0
2 Almassora 1.8 Borriana 4.1
```

La clase **ListGraph<T,W>** almacena un grafo dirigido con arcos con peso usando listas de adyacencia. La parte privada de su implementación es:

```
public class ListGraph<T,W> implements Graph<T, W> {
    private class Node<T> {
        T data;
        List<EDEdge<W>> lEdges;

        Node (T data) {
            this.data = data;
            this.lEdges = new LinkedList<EDEdge<W>>();
        }
    }

    private boolean isDirected;
    private List<Node> nodes = new ArrayList<Node>();
    ...
}
```

Implementar constructor de la clase **ListGraph<String,Double>** que reciba como parámetro el nombre del fichero y construya el grafo correspondiente.

```
    public ListGraph (String nomFich);
```

Curso 2012/13 Examen Final 1ª Convocatoria

4. 12 **(2.5 puntos)** Un grafo no dirigido puede utilizarse para representar un laberinto, donde cada nodo representa una encrucijada y cada arco un camino. Dado un grafo representado un laberinto, implementado con listas de adyacencia, y el nodo entrada (**start**) y el nodo salida (**exit**), escribe una operación que calcule un camino entre la entrada y salida. Devolverá un vector de caminos. Cada elemento del este vector representará a un nodo del grafo según su índice, y almacenará el índice del nodo desde el que se ha llegado a él, o -1 si el nodo no ha sido visitado. El algoritmo debe ser eficiente en el sentido que se valorará que una vez haya llegado a la salida no continúe buscando caminos.

```
public int[] findPath (int start, int exit);
```

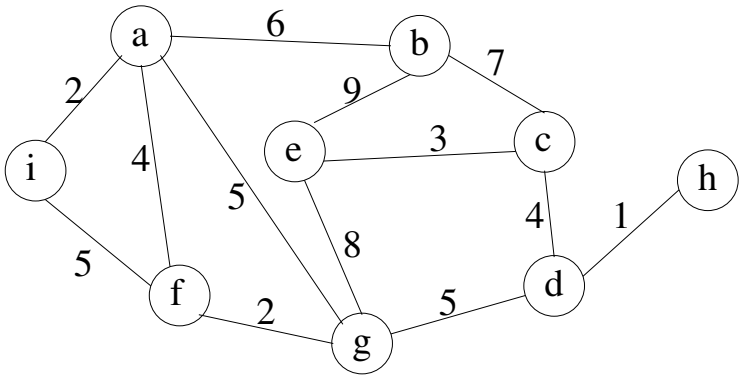
La parte privada de la clase es:

```
public class EDListGraph<T, W> implements EDGraph<T, W> {
    private class Node<T> {
        public T data;
        public List<EDEdge<W>> lEdges;
    }

    private boolean isDirected;
    private ArrayList<Node<T>> nodes;
...
}
```

Curso 2013/14 Examen Final 2ª Convocatoria

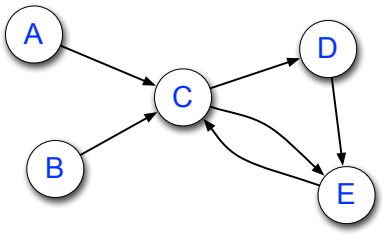
4. 13 **(1 punto)** Dado el siguiente grafo realizar un recorrido en anchura desde el nodo etiquetado con la letra *d*. La respuesta debe mostrar claramente la traza del algoritmo de recorrido en anchura y debe quedar claro qué nodos se visitan en cada iteración.



Curso 2012/13 Examen Final 2ª Convocatoria

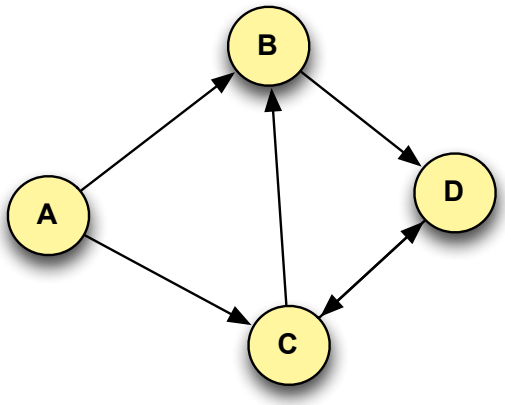
4. 14 **(2 puntos)** Dado el grafo de la figura:

1. Realiza un recorrido en profundidad empezando en el nodo etiquetado con la letra 'a'. Indicar claramente la secuencia de nodos visitados.
2. Escribid la matriz de adyacencia que representa el grafo.
3. Escribid las listas de adyacencia para los nodos etiquetados con las letras 'A', 'C' y 'D'.



Curso 2011/12 Examen Final 1ª Convocatoria

4. 15 **(2 puntos)** Pretendemos implementar un grafo con arcos sin peso usando un diccionario. De esta forma un grafo con nodos de la clase **T** se implementa como un diccionario en el que cada par se compone de un elemento de clase **T** y un lista de elementos de clase de **T**. Los elementos de la lista son los nodos adyacentes desde el nodo correspondiente. Como ejemplo el grafo de la figura se describiría con los siguientes pares: **(A, [B, C])**, **(B, [D])**, **(C, [B, D])**, **(D, [C])**.



Escribe una función estática que dado un grafo, y dos nodos concretos, uno inicial y otro final, determine si existe un camino que los una. *Pista: usad un conjunto de nodos que inicialmente contenga el primer elemento, e id sucesivamente añadiendo los elementos a los que se puede llegar desde cada uno de los nodos del conjunto.*

Curso 2011/12 Examen Final 1ª Convocatoria