Grafos

- Introducción a los grafos
- Especificación del TAD Grafo
- Implementación del TAD Grafo, ilustrando las principales estructuras de datos usadas para esta abstracción.
- Principales algoritmos de grafos

Grafos

Contenidos

1. **Terminología fundamental**
2. **TAD grafo**
   a) **Especificación**
   b) **Implementación**
      - **Mediante listas de arcos**
      - **Mediante listas de adyacencia**
      - **Mediante matriz de adyacencia**

Grafos

- **Bibliografía**
  - Goodrich M. y Tamassia R., *Data structures and Algorithms in JAVA 4ª ed.* John Wiley & Sons Inc. , 2006 Págs.580-592
  - Goldman S. y Goldman K. *A practical guide to data structures and algorithms using Java*. Chapman & Hall/CRC. 2008. Págs.843-856
  - Allen Weiss, M. *Estructuras de datos en java*. 4ª ed. Pearson, 2013 Págs.515-557

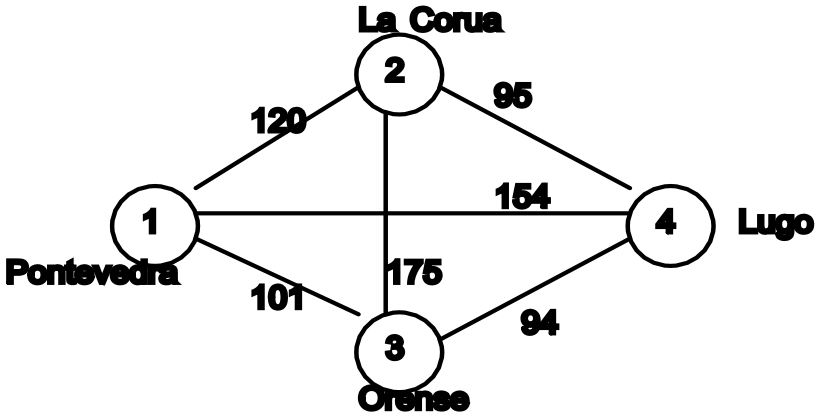
TAD Grafo

- Un grafo es una estructura capaz de representar relaciones complejas entre objetos de un mismo tipo. Formalmente se representa mediante el par G = (V, A), donde:
  - V es un conjunto de objetos llamados **vértices o nodos** (*Un vértice es un objeto compuesto de una etiqueta*)
  - A es un conjunto de objetos denominados **aristas o arcos.** Las aristas representan relaciones entre los vértices, de forma que una arista es un par (u, v) de vértices de V, y opcionalmente una etiqueta.

- Básicamente, podemos clasificar los grafos en 4 tipos dependiendo de dos criterios:
  - Grafo dirigido. Es aquel cuyas aristas forman pares ordenados.
  - Grafo no dirigido. Es aquel cuyas aristas no son pares ordenados.
  - Grafo etiquetado o valorado. Cuando se asocia información a cada arista de un grafo
  - Grafo no etiquetado. Cuando no se asocia ninguna información a las aristas

TAD Grafo

- Un ejemplo de grafo podría ser la red de carreteras de Galicia, donde los vértices representan las ciudades y las aristas la carretera que une dos ciudades:



TAD Grafo

- Algunas definiciones sobre grafos:
  - Un **camino** en un grafo G = (V, A) es una secuencia de vértices $v_1, ..., v_n \in V$, con n>=1, tal que $(v_{i-1}, v_i) \in A$ para i= 1...n-1
  - La **longitud** de un camino es su número de vértices menos 1
  - Un camino es **simple** si todos sus vértices, excepto tal vez el primero y el último, son distintos
  - Un **ciclo** es un camino simple de longitud no nula que empieza y termina en el mismo vértice
  - En un grafo no dirigido, el **grado** de un vértice *v* es el número de aristas que contiene a *v*
  - Se dice que el vértice y es **sucesor** o **adyacente** del vértice x si existe una arista que tenga por origen a *x* y por destino a *y*, es decir, si la arista $(x, y) \in A$
  - Se dice que el vértice x es **antecesor** del vértice *y* si existe una arista que tenga por origen a *x* y por destino a *y*, es decir, si la arista $(x, y) \in A$

TAD Grafo

- En un grafo dirigido se distingue entre el **grado de entrada** y el **grado de salida**: el grado de entrada de un vértice *v* es el número de aristas que llegan a *v* (antecesores), y el grado de salida de un vértice *v* es el número de aristas que salen de *v* (sucesores).

- Un grafo no dirigido G es **conexo** si existe un camino entre cualquier par de nodos que forman el grafo.

- Un grafo dirigido es **fuertemente conexo** si existe un camino entre cualquier par de nodos que forman el grafo.

TAD Grafo

Especificación

```
public class Grafo <E,T>
    // características: Es un conjunto de vértices y un conjunto de arcos. Los objetos son modificables.
    public Grafo( )
        // Produce:un grafo vacío
    public boolean estaVertice(Vertice<E> v)
        // Produce:        cierto si el vértice v está en this, falso en caso contrario
    public boolean estaArco(Arco<E,T> a)
        // Produce:        cierto si el arco a está en this, falso en caso contrario
    public Iterator<Vertice<E>> vertices()
        // Produce:        devuelve un iterador sobre los vértices del grafo
    public Iterator<Arco<E,T>> arcos();
        // Produce:        devuelve un iterador sobre los arcos del grafo
    public Iterator<Vertice<E>> adyacentes (Vertice<E> v)
        // Produce:        devuelve un iterador sobre los vértices del grafo adyacentes al
        //                 vértice v
```

TAD Grafo

```
    public void insertarVertice (Vertice<E> v)
        // Modifica:        this
        // Produce:         inserta el vértice v en this
    public void insertarArco (Arco<E,T> a)
        // Modifica:        this
        // Produce:         inserta el arco a  en this
    public void eliminarVertice (Vertice<E> v)
        // Modifica:        this
        // Produce:         elimina el vértice v de this y todos los arcos
        //                  que salen y llegan a v
    public void eliminarArco (Arco<E,T> a)
        // Modifica:        this
        // Produce:         elimina el arco a de this
}
```

TAD Grafo

Ejemplo de uso del TAD Grafo

```
public static <E,T> int gradoSalida (Grafo<E,T> g, Vertice<E> v){
  Iterator<Vertice<E>> adys = g.adyacentes(v);
  int gs = 0;
  while (adys.hasNext()) {
            adys.next ();
            gs++;
  }
  return gs;
}
```

TAD Grafo

Ejemplo de uso del TAD Grafo

```
public static <E,T> Iterator<Vertice<E>> predecesores (Grafo <E,T> g, Vertice<E> v)
        Vector<Vertice<E>> pred = new Vector<>();
        Iterator<Vertice <E>> itv = g.vertices();
        while (itv.hasNext()){
                Vertice<E> w = itv.next();
                Iterator<Vertice<E>> it2 = g.adyacentes(w);
                while (it2.hasNext()){
                        if (it2.next().equals(v))
                        pred.add(w);
                 }
        }
        return pred.iterator();
}
```

Algoritmos de recorrido

- La operación de recorrer un grafo consiste en partir de un vértice determinado y visitar todos aquellos vértices que son accesibles desde él en un determinado orden.
- Dos estrategias:
  - recorrido en profundidad (*DFS: Depth First Search*)
  - recorrido en anchura (*BFS: Breadh First Search*).

Algoritmos de recorrido

Recorrido en profundidad

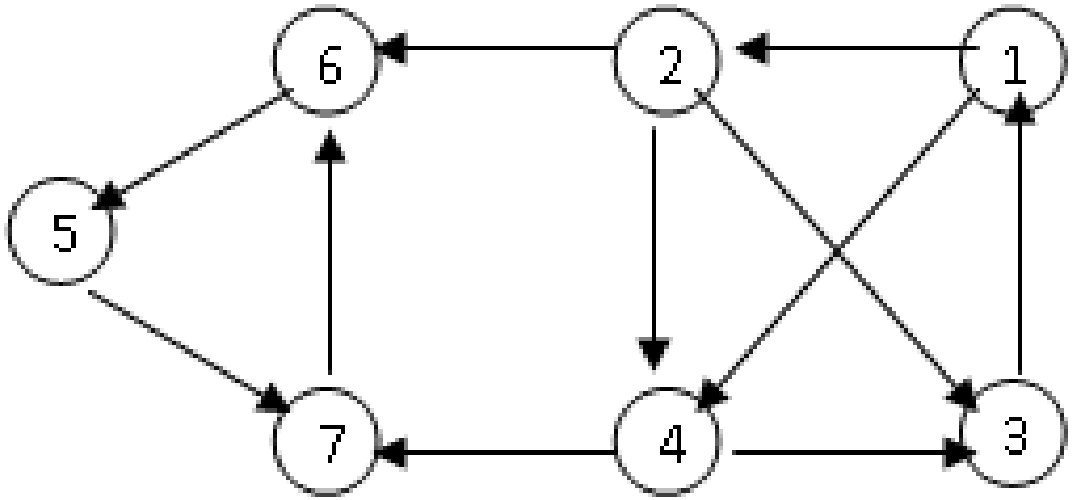
- Es una generalización del recorrido en *preorden* de un árbol.
- Pasos:
  1. Se visita el vértice del que se parte, **v**.
  2. Se selecciona un vértice **w**, adyacente a **v** y que todavía no se haya visitado.
  3. Se realiza el recorrido en profundidad partiendo del vértice **w**.
  4. Cuando se encuentra un vértice cuyo conjunto de adyacentes han sido visitados en su totalidad se retrocede hasta el último vértice visitado que tenga vértices adyacentes no visitados y se ejecuta el paso 2.
- Suponemos la existencia de un conjunto (*visitados*) para ir almacenando los vértices del grafo por los que se va pasando.
- En principio el conjunto de vértices visitados estará vacío.

Algoritmos de recorrido

```
public static <E,T> void profundidad (Grafo<E,T> g, Vertice<E> v) {
        Vector<Vertice<E>> visitados = new Vector<>();
        profundidad(g, v, visitados);
}

private static <E,T> void profundidad(Grafo<E,T> g, Vertice<E> v,
                                      Vector<Vertice<E>> visitados)
{
        System.out.println(v);
        visitados.add(v);
        Iterator<Vertice<E>> adys = g.adyacentes(v);
        while (adys.hasNext()){
            Vertice<E> w = adys.next();
            if (!visitados.contains(w))
                profundidad(g, w, visitados);
        }
}
```

Algoritmos de recorrido



Recorrido en profundidad: **1 2 3 4 7 6 5**

Algoritmos de recorrido

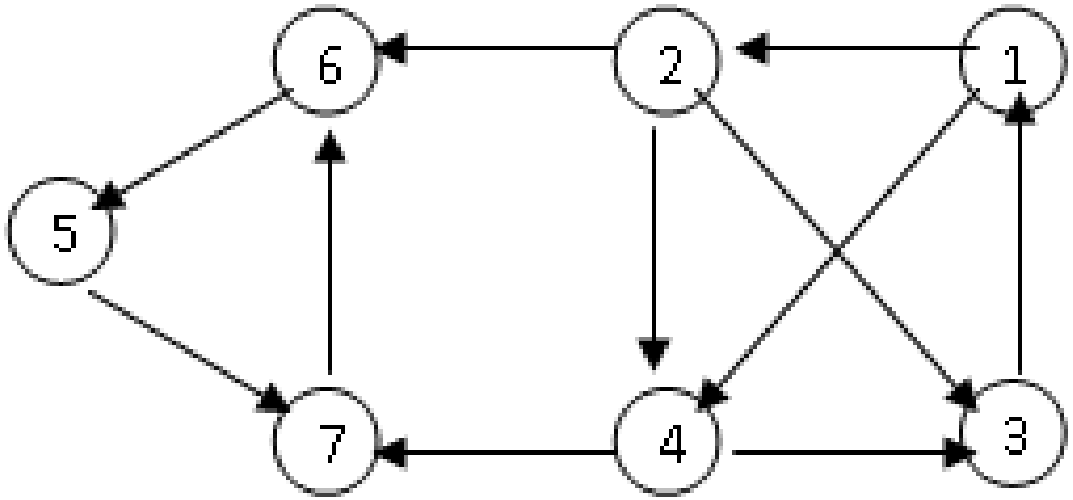
Recorrido en anchura

- Generaliza el recorrido en *anchura* (por niveles) de un árbol.
- Pasos:
  1. Se visita el vértice del que se parte, **v**.
  2. Se visitan todos sus adyacentes que no estuvieran ya visitados, y así sucesivamente. Esto es, se visitan todos los vértices adyacentes antes de pasar a otro vértice.
- Utilizamos un conjunto (*visitados*) para ir almacenando los vértices del grafo por los que se va pasando.
- En una cola (*porExplorar*) se mantienen los vértices adyacentes que se han obtenido a partir de los vértices visitados y cuyos adyacentes aún restan por explorar.
- Inicialmente, tanto el conjunto de vértices visitados como la cola de vértices por visitar estarán vacíos.

Algoritmos de recorrido

```
public static <E,T> void anchura(Grafo<E,T> g, Vertice<E> v){
        Vector<Vertice<E>> visitados = new Vector<>();
        Cola<Vertice<E>> porExplorar = new EnlazadaCola<>();
        porExplorar.insertar(v);
        visitados.add(v);
        do {
            v = porExplorar.suprimir();
            System.out.println(v);
            Iterator<Vertice<E>> adys = g.adyacentes(v);
            while (adys.hasNext()) {
                Vertice<E> w = adys.next();
                if (!visitados.contains(w)) {
                    porExplorar.insertar(w);
                    visitados.add(w);
                }
            }
        }
        while (!porExplorar.esVacio());
    }
```

Algoritmos de recorrido



Recorrido en anchura: **1 2 4 3 6 7 5**

Implementación TAD Grafo

Implementación

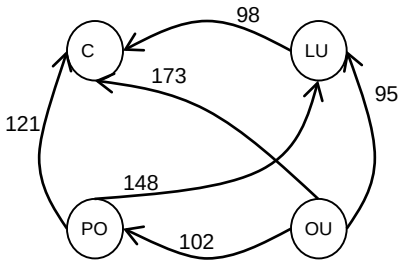
- Paso 1: Definición interfaz

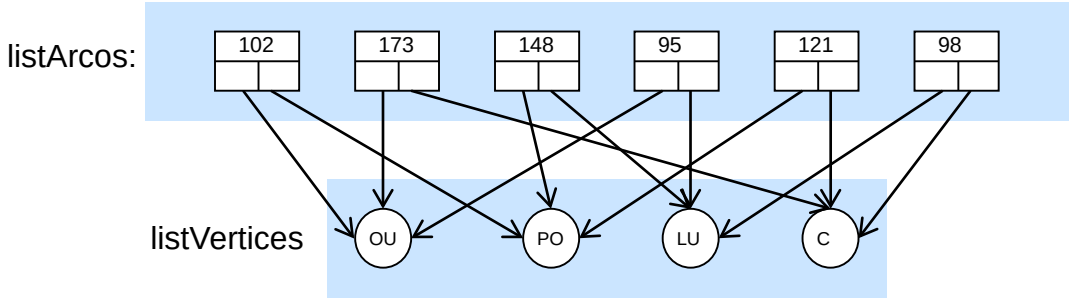
```
public interface Grafo<E,T>{
        public boolean esVacio();
        public boolean estaVertice(Vertice<E> v);
        public boolean estaArco(Arco<E,T> a);
        public Iterator<Vertice<E>> vertices();
        public Iterator<Arco<E,T>> arcos();
        public Iterator<Vertice<E>> adyacentes (Vertice<E> v);
        public void insertarVertice (Vertice<E> v);
        public void insertarArco (Arco<E,T> a);
        public void eliminarVertice (Vertice<E> v);
        public void eliminarArco (Arco<E,T> a);
}
```

- Paso 2: Clase implemente la interfaz
  - Mediante estructuras genéricas
    - public class ListaDeArcos<E,T> implements Grafo<E,T>
    - public class ListaDeAdyacencia<E,T> implements Grafo<E,T>
    - public class MatrizDeAdyacencia<E,T> implements Grafo<E,T>

Implementación TAD Grafo

- Mediante una lista de arcos
  - Representación






```
public class ListaDeArcos<E,T> implements Grafo<E,T>{
    private Lista<Vertice<E>> listVertices;
    private Lista<Arco<E,T>> listArcos;
```

Implementación TAD Grafo

```
public class Vertice<E>{

    private E etiqueta;

    // métodos
    public Vertice(E etiqueta)  {…}
    public E getEtiqueta()  {…}
    public void setEtiqueta(E etiqueta) {...}
}
```

```
public class Arco <E,T>{

    private Vertice<E> origen;
    private Vertice<E> destino;
    private T etiqueta;

    //métodos
    public Arco(Vertice<E> vo, Vertice<E> vd, T etiq) {…}
    public Vertice<E> getOrigen() {…}
    public Vertice<E> getDestino() {…}
    public T getEtiqueta() {…}
    public void setOrigen(Vertice<E> vo) {…}
    public void setDestino (Vertice<E> vd) {…}
    public void setEtiqueta(T etiq) {…}
}
```

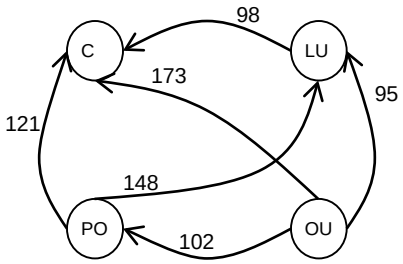
Implementación TAD Grafo

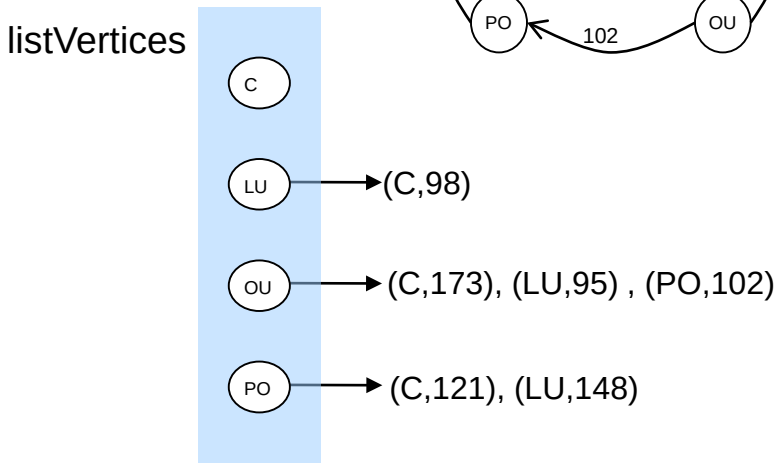
Una posible implementación del método adyacentes del TAD:

```
public Iterator<Vertice<E>> adyacentes (Vertice<E> v){
        Vector<Vertice<E>> ady = new Vector<>();
        for (Arco<E,T> arc: listArcos)
                if (arc.getOrigen().equals(v))
                        ady.add(arc.getDestino());
        return ady.iterator();
   }
```

Implementación TAD Grafo

- Mediante una lista de adyacencia
  - Representación






```
public class ListaDeAdyacencia<E,T> implements Grafo<E,T>{
    private Lista<VerticeConLista<E,T>> listVertices;
    private int numVertices;
```

Implementación TAD Grafo

```
public class Vertice<E>{

    private E etiqueta;

    // métodos
    public Vertice(E etiqueta)  {…}
    public E getEtiqueta()  {…}
    public void setEtiqueta(E etiqueta) {...}
}
```

```
public class Arco <E,T>{

    private Vertice<E> origen;
    private Vertice<E> destino;
    private T etiqueta;

    //métodos
    public Arco(Vertice<E> vo, Vertice<E> vd, T etiq) {…}
    public Vertice<E> getOrigen() {…}
    public Vertice<E> getDestino() {…}
    public T getEtiqueta() {…}
    public void setOrigen(Vertice<E> vo) {…}
    public void setDestino (Vertice<E> vd) {…}
    public void setEtiqueta(T etiq) {…}
}
```

Implementación TAD Grafo

```
public class VerticeConLista <E,T>
    private Vertice<E> verticeOrigen;
    private Lista<Adyacente<E,T>> listAdyacentes;
    //métodos
    public VerticeConLista (Vertice<E> verticeOrigen, Lista<VerticeAdyacente<E,T>> ady)
    {…}
    public Vertice<E> getVertice() {…}
    public Lista<Adyacente<E,T>> getLista{…}
}

public class VerticeAdyacente<E,T> {
    private Vertice<E> verticeDestino;
    private T etiquetaArco;
    // métodos
    public VerticeAdyacente(Vertice<E> verticeDestino, E etiquetaArco) {...}
    public Vertice<E> getDestino() {...}
    public getEtiqueta() {... }
}
```

Implementación TAD Grafo

Una posible implementación del método adyacentes:

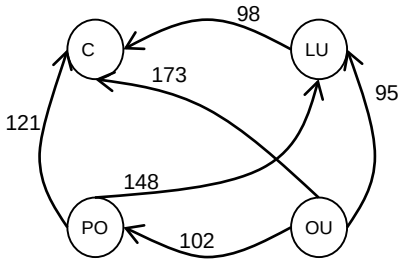
```
public Iterator<Vertice<E>> adyacentes (Vertice<E> v){

    Vector<Vertice<E>> ady = new Vector<>();
    for (VerticeConLista<E,T> w: listVertices)
            if(w.getVertice().equals(v)){
                    Lista<VerticeAdyacente<E,T>> l = w.getLista();
                    for (Adyacente<E,T> q : l)
                            ady.add(q.getDestino());
             }

    return ady.iterator();
```

Implementación TAD Grafo

- Mediante una matriz de adyacencia
  - Representación



vertices

0	C
1	LU
2	OU
3	PO

matAdy

	0	1	2	3
0	—	—	—	—
1	98	—	—	—
2	173	95	—	102
3	121	148	—	—

```
public class MatrizDeAdyacencia<E,T> implements Grafo<E,T>{
    private Vertice<E> [ ] vertices;
    private T [][] matAdy;
    private int numVertices;
```

Implementación TAD Grafo

Una posible implementación del método adyacentes:

```
private int posicionVertice (Vertice<E> v){
        for (int i=0;i< numVertices; i++)
          if (vertices[i].equals(v))
             return i;
       return -1;
}
public Iterator<Vertice<E>> adyacentes (Vertice<E> v){
        Vector<Vertice<E>> vertAdys = new Vector<>();
        if (estaVertice(v)){
          int ivo = posicionVertice(v);
          for (int ivd = 0; ivd< numVertices; ivd++)
             if (matAdy[ivo][ivd] != null)
                vertAdys.add(vertices[ivd]);
       }
       return vertAdys.iterator();
    }
```

Uso TAD Grafo

Realizar la actividad 8.