Grafos

Introducción

- Multitud de aplicaciones prácticas se pueden modelar de forma sencilla por medio de entidades y relaciones entre estas entidades. E.g.
  - Redes (Datos, Energía, Teleco, etc.)
  - Mapas (GPS)
  - Redes sociales
  - Web
  - Comercio y finanzas

Concepto de grafo

- Un grafo es una abstración compuesta por
  - Vértices (nodos – *vertex/node*)
  - Aristas (enlaces – *links/edges*)
- La teoría de grafos estudia las propiedades de estos grafos y algoritmos para resolver problemas de grafos, los cuales tienen mucha utilidad práctica en las aplicaciones concretas.

Ejemplos de aplicación

- Encontrar la ruta más corta en un grafo : GPS, Enrutamiento en Internet.
- Encontrar un árbol de cubrimiento mínimo : Redes LAN Ethernet, diseño de redes de distribución (agua, energía).
- Encontrar nodos populares: Rankeo de páginas web en búscadores.

Tipos de grafos

Según la forma de las aristas se distinguen:

- **No dirigidos**: Las aristas no tienen un sentido. Se pueden recorrer en ambas direcciones. (e.g. Una conexión a la red)
- **Dirigidos**: Las aristas tienen una dirección asignada. Solo se pueden recorrer en el sentido de la arista. (e.g. Un link en una página web)

Grafos no dirigidos

- Son colecciones de vértices y aristas no dirigidas.
- Los vértices se identifican por números 0..V-1
- Las aristas se denotan por los **pares no ordenados** de los vértices que conectan, e.g. {a,b}
- Es posible tener bucles (self-loops) y aristas paralelas. Inicialmente no se consideran para simplificar.

$$G = \langle V, E \rangle$$

Grafos dirigidos

- Son colecciones de vértices y aristas no dirigidas.
- Los vértices se identifican por números 0..V-1
- Las aristas se denotan por los **pares ordenados** de los vértices que conectan, e.g. (a,b)
- Es posible tener bucles (self-loops) y aristas paralelas. Inicialmente no se consideran para simplificar.

$$G = <V,E>$$

Terminología

- **Camino**: Es una secuencia de vértices conectados por aristas. Es simple si no se repiten vértices. Es un ciclo si el primer y el último vértices coinciden.
- **Conexo**: Un grafo es conexo si existe un camino entre todo par de nodos del grafo. En caso contrario es **no conexo** y se identifican múltiples componentes conexas del grafo.

Terminología

- **Acíclico**: Es un grafo que no contiene ciclos. Por ejemplo todo árbol es un grafo conexo y acíclico y viceversa.
- **Grafo dirigido acíclico (DAG)**: Un gráfo dirigido libre de ciclos.

Terminología

- **Grado de un nodo:** Es el número de aristas que se conectan al nodo. Alternativamente, es el número de nodos *adyacentes* al nodo.
- **Clique**: Se le llama así a un grafo completo, todas las aristas posibles se encuentran en el grafo.

Densidad de un grafo

- Hace referencia al número de aristas del grafo con respecto al número de nodos.
- Se dice que el grafo es no denso (o disperso) si el número de aristas se encuentra dentro de un factor constante del número de vértices:

$$E \leq cV$$

- En caso contrario, se dice que el grafo es denso.

Grafos bipartitos

- Cuando los vértices del grafo se pueden descomponer en dos conjuntos, de forma tal que toda arista conecta un vértice de un conjunto con un vértice del otro conjunto.

Grafo no dirigido como un tipo de dato abstracto

class Graph		
	Graph(int V)	// Constructor indicando número de vértices
	Graph(In in)	// Constructor utilizando un inputStream
int	V()	// Número de vértices
int	E()	// Número de aristas
	addEdge(int u, int v)	// Adicionar una arista
Iterable<Integer>	adj(int v)	// Determinar los nodos adyacentes a v
int	degree(int v)	// Grado del nodo
String	toString()	// Representación textual del grafo

Ver implementación

Grafo dirigido como un tipo de dato abstracto

class DiGraph		
	`Digraph(int V)`	// Constructor indicando número de vértices
	`Digraph(In in)`	// Constructor utilizando un inputStream
`int`	`V()`	// Número de vértices
`int`	`E()`	// Número de aristas
	`addEdge(int u, int v)`	// Adicionar una arista
`Iterable<Integer>`	`adj(int v)`	// Determinar los nodos adyacentes a v
`int`	`outdegree(int v)`	// Grado saliente del nodo
`int`	`indegree(int v)`	// Grado entrante del nodo
`String`	`toString()`	// Representación textual del grafo
Digraph	`reverse()`	// Grafo con aristas invertidas

Ver implementación

Representación de un grafo

Existen diversas formas, las dos más comunes:

- Matrix de adyacencia: A_{ij} es cero si no hay arista i-j, o uno en caso contrario.
- Listas de adyacencia: Se construyen V listas, cada una con los vértices adyacentes a cada nodo. Las listas se almacenan en un vector indexado por el id del nodo de partida.

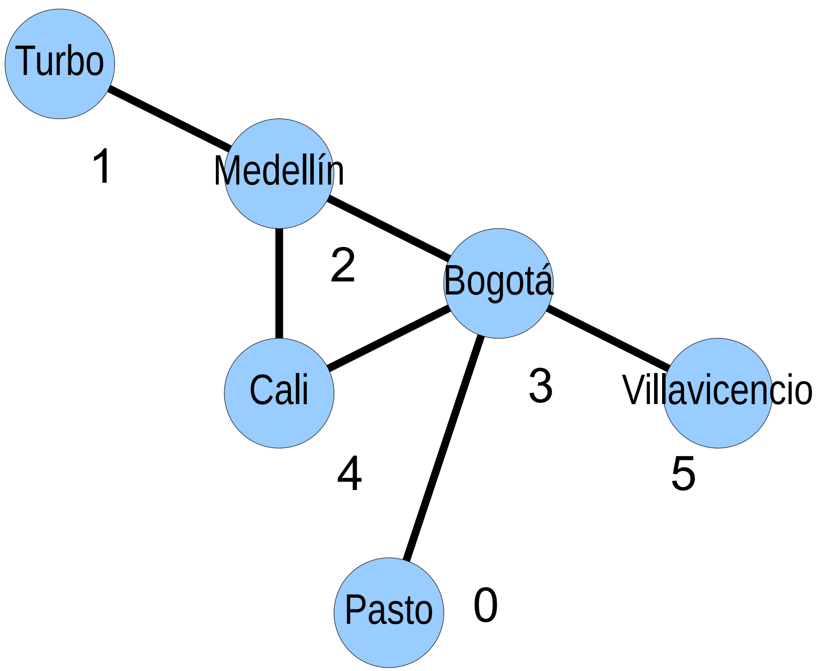
Ejercicio: Comparar la complejidad espacial de ambas representaciones.

Implementación del grafo no dirigido (Listas de adyacencia)

```
public class Graph {
    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    public Graph(int V) {
        if (V < 0) throw new IllegalArgumentException("Number of vertices must be nonnegative");
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    public int V() { return V; }
    public int E() { return E; }

    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int degree(int v) {
        return adj[v].size();
    }


}
```

Ver la implementación completa

Ejemplo: Grafo no dirigido




```
Graph vuelos = new Graph(6);
vuelos.addEdge(1, 2);
vuelos.addEdge(2, 3);
vuelos.addEdge(2, 4);
vuelos.addEdge(3, 5);
vuelos.addEdge(3, 0);


StdOut.println(vuelos.degree(3));


for(int vecino: vuelos.adj(2)) {
    StdOut.print(vecino+", ");
}
StdOut.println();


StdOut.println(vuelos);
```

Ejercicio

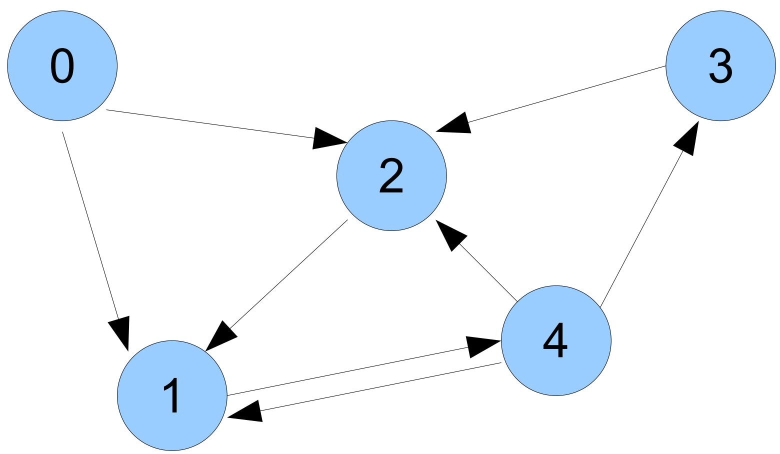
- Cómo borrar una arista?

Implementación del grafo dirigido

```
public class Digraph {
    private final int V;
    private int E;
    private Bag<Integer>[] adj;
    private int[] indegree;


    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
        indegree[w]++;
        E++;
    }

    public int outdegree(int v) {
        return adj[v].size();
    }

    public int indegree(int v) {
        return indegree[v];
    }


}
```

Ver la implementación completa

Ejemplo: Grafo dirigido




```
Digraph g = new Digraph(5);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(2, 1);
g.addEdge(1, 4);
g.addEdge(3, 2);
g.addEdge(4, 1);
g.addEdge(4, 2);
g.addEdge(4, 3);

StdOut.println(g);

StdOut.println(g.indegree(4));
StdOut.println(g.outdegree(4));

for(int vecino: g.adj(4))
    StdOut.println("Vecino de 4: "+vecino);

Digraph ginv = g.reverse();
StdOut.println(ginv);
```

Ejercicio

- Se quiere implementar un Graph o Digraph nombre se indiquen los nodos por medio de una etiqueta.

Solución

```
public class DigrafoConNombres {

    private Map<String,Integer> nombres = new HashMap<>();
    private Map<Integer,String> numeros = new HashMap<>();
    private Digraph graph;

    public DigrafoConNombres(String[] nombres) { … }

    public void addEdge(String a, String b) throws Exception { … }

    public Digraph getDigraph() { … }

    public String getNombre(int n) { … }

    public String toString() { … }

}
```

Búsquedas en grafos
Recorrido de grafos

- Hay aplicaciones en las que se requieren operaciones tales como:
  - Visitar todos los nodos del grafo una vez
  - Encontrar un camino en el grafo
  - Determinar si el grafo es conexo
- Este tipo de problemas se resuelven por medio de algoritmos de recorrido de grafos.

Tipos de recorrido

- **Recorrido en profundidad (Depth First Search – DFS)** : Se visita un nodo, se marca y se visitan recursivamente todos sus adyacentes aún no visitados.
- **Recorrido en anchura (Breath First Search – BFS)**: Al visitar un nodo se marca, luego se visitan todos sus adyacentes y solo después de esto se visitan los vecinos de los adyacentes.

Recorrido en profundidad (DFS) Versión recursiva

```
public class DepthFirstSearch {
    private boolean[] marked;    // marked[v] = is there an s-v path?
    private int count;           // number of vertices connected to s

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v) {
        count++;
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

}
```

Ver implementación completa

Implementando el patrón de diseño Visitor

- Visitor es un patrón de diseño que permite separar el algoritmo de las operaciones que se realizan sobre objetos/estructuras que se operan.

```
public interface Visitor<T> {
    public void visit(T x);
}



private Visitor<Integer> visitor;

private void dfs(Graph G, int v) {
    count++;
    marked[v] = true;
    if (visitor!=null) visitor.visit(v);
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

Recorrido en profundidad (DFS) Versión no recursiva

```
public class NonrecursiveDFS  {
      private boolean[] marked;    // marked[v] = is there an s-v path?


      public NonrecursiveDFS(Graph G, int s) {
        marked = new boolean[G.V()];

        validateVertex(s);

        Iterator<Integer>[] adj = (Iterator<Integer>[]) new Iterator[G.V()];
        for (int v = 0; v < G.V(); v++)
            adj[v] = G.adj(v).iterator();

        // depth-first search using an explicit stack
        Stack<Integer> stack = new Stack<Integer>();
        marked[s] = true;
        stack.push(s);
        while (!stack.isEmpty()) {
            int v = stack.peek();
            if (adj[v].hasNext()) {
                int w = adj[v].next();
                // StdOut.printf("check %d\n", w);
                if (!marked[w]) {
                    marked[w] = true;
                    stack.push(w);
                    // StdOut.printf("dfs(%d)\n", w);
                }
            }
            else {
                // StdOut.printf("%d done\n", v);
                stack.pop();
            }
        }
    }

}
```

Implementación completa

Ejercicio

- Implementar el patrón Visitor en el recorrido DFS no recursivo.

Recorrido en anchura (Breath First Search - BFS)

```
public class RecorridoBFS {


     // Se asume que el grafo es conexo
     public static void BFS(Graph g, int s) {
         Queue<Integer> paraVisitar = new Queue<>();
         boolean[] marcados = new boolean[g.V()];
         paraVisitar.enqueue(s);
         marcados[s]=true;

         while(!paraVisitar.isEmpty()) {
             Integer actual = paraVisitar.dequeue();
             StdOut.println("Visitado: "+actual);
             for(Integer vecino: g.adj(actual)) {
                 if (!marcados[vecino]) {
                     marcados[vecino]=true;
                     paraVisitar.enqueue(vecino);
                 }
             }

         }

     }

     public static void main(String[] args) {
         Graph g = GraphGenerator.simple(10,20);
         BFS(g,0);
     }


}
```

Ejemplos de aplicación de los recorridos

1) Determinar si el grafo es conexo
2) Determinar cuantas componentes conexas tiene un grafo
3) Determinar si un grafo contiene ciclos
4) Búsquedas exhaustivas:
   - Resolver un laberinto
   - Juegos de turnos: Ajedrez, damas

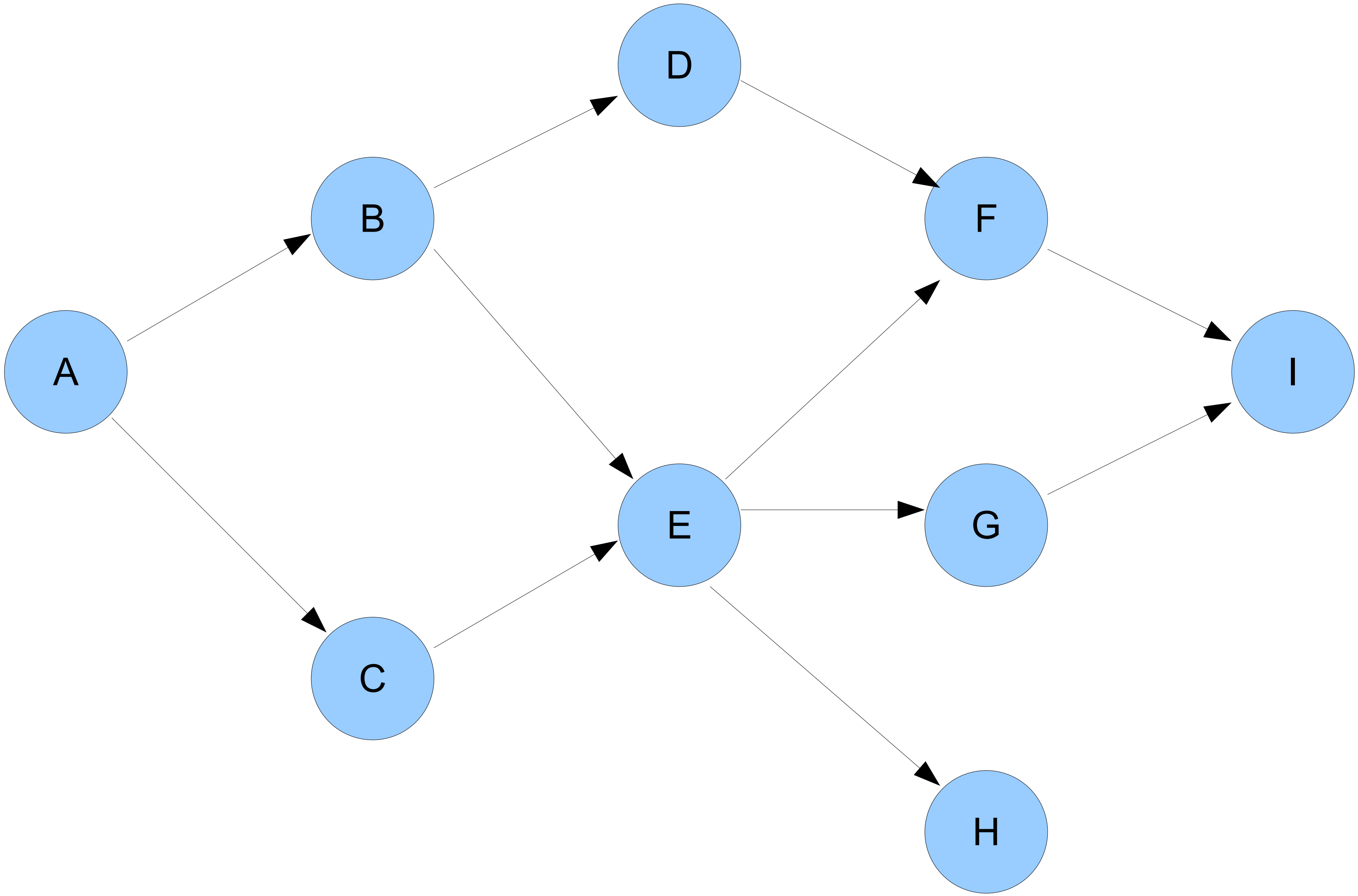
Ordenamiento topológico

- Consideremos una situación donde se tienen una serie de tareas que tienen dependencias unas de otras, por ejemplo:
  - Compilar un programa y sus módulos
  - Ejecutar un proyecto
  - Cursos en la malla curricular
- El problema es encontrar un orden <u>factible</u> para ejecutar las distintas tareas.

Consideraciones para el ordeamiento topológico

- El grafo debe ser un DAG
- Un recorrido en profundidad permite ordenar los nodos del grafo tanto en pre-orden como en post-orden.
- Si enumeramos los nodos en post-orden, un nodo solo se enumera después de haber visitado todas sus dependencias.
- Si enumeramos los nodos en orden inverso del post-orden, se tienen un ordenamiento topológico.

Ejemplo



Implementación de la solución (1)

- Se implementa un recorrido DFS que guarde registro del post-orden

```
public class DepthFirstOrder {
    private boolean[] marked;
    private int[] pre;
    private int[] post;
    private Queue<Integer> preorder;
    private Queue<Integer> postorder;
    private int preCounter;
    private int postCounter;

    public DepthFirstOrder(Digraph G) {
        pre    = new int[G.V()];
        post   = new int[G.V()];
        postorder = new Queue<Integer>();
        preorder  = new Queue<Integer>();
        marked    = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);

    }
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        pre[v] = preCounter++;
        preorder.enqueue(v);
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        post[v] = postCounter++;
    }
```

Ver DepthFirstOrder.java

Implementación de la solución (2)

- Luego de hacer el recorrido se invierte el orden de los nodos en el post-orden

```
public class Topological {
    private Iterable<Integer> order;
    private int[] rank;

    public Topological(Digraph G) {
        DirectedCycle finder = new DirectedCycle(G);
        if (!finder.hasCycle()) {
            DepthFirstOrder dfs = new
DepthFirstOrder(G);
            order = dfs.reversePost();
            rank = new int[G.V()];
            int i = 0;
            for (int v : order)
                rank[v] = i++;
        }
    }
…

public class DepthFirstOrder {

    public Iterable<Integer> reversePost() {
        Stack<Integer> reverse = new Stack<Integer>();
        for (int v : postorder)
            reverse.push(v);
        return reverse;
    }
```

Ver Topological.java

Ejemplo

```
public static void main(String[] args) throws Exception {
    String[] nombres = {"A","B","C","D","E","F","G","H","I"};
    DigrafoConNombres g = new DigrafoConNombres(nombres);
    g.addEdge("A", "B");
    g.addEdge("A", "C");
    g.addEdge("B", "D");
    g.addEdge("B", "E");
    g.addEdge("C", "E");
    g.addEdge("D", "F");
    g.addEdge("E", "F");
    g.addEdge("E", "G");
    g.addEdge("E", "H");
    g.addEdge("F", "I");
    g.addEdge("G", "I");

    StdOut.println(g);

    Topological topological = new Topological(g.getDigraph());
    for(int i: topological.order()) {
        StdOut.println(topological.rank(i)+" : "+g.getNombre(i));
    }

}
```