

Grafos: Implementaciones y recorridos básicos

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025

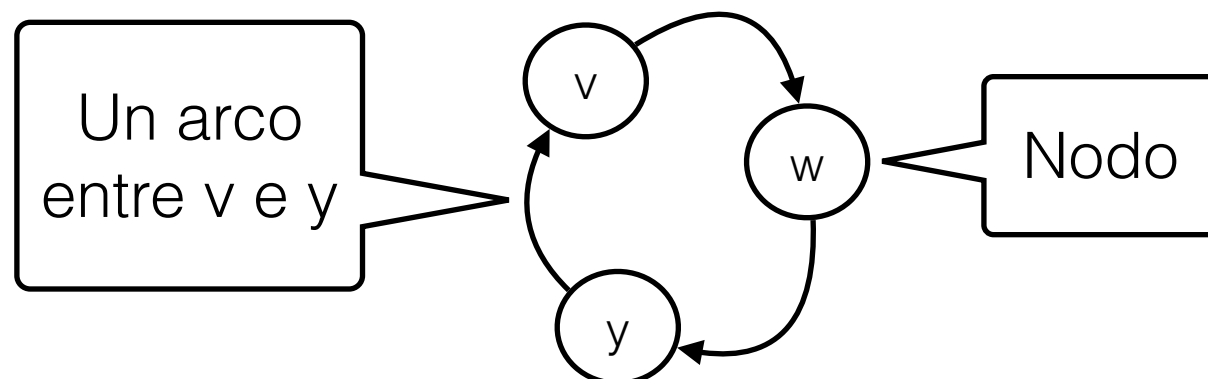
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET



Grafos

Los grafos contienen dos tipos de elementos:

- **Nodos** o vértices: generalmente guardan algún tipo de información,
- **Arcos**: son conexiones entre vértices, también pueden contener información.
 - Grafos simples: Hay a lo sumo un arco entre cada par de nodos
 - Multigrafos: Admiten más de un arco entre un mismo par de nodos



Trabajaremos con grafos simples en la materia

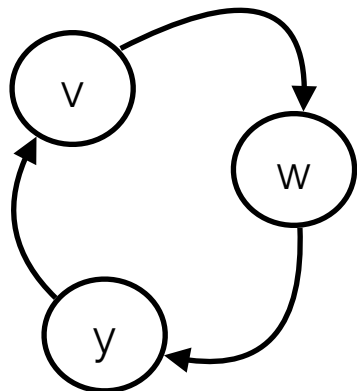
Definición

Los grafos se pueden definir formalmente de la siguiente forma:

Un grafo es un par (V, E) en donde:

- V es un conjunto de nodos,
- $E \subseteq V \times V$ es una relación.

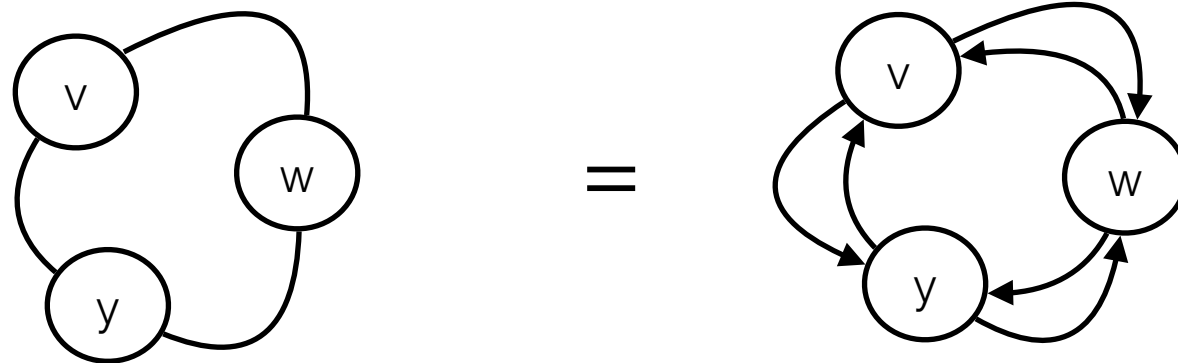
Por ejemplo:



$$= (\{v, w, y\}, \{(v, w), (w, y), (y, v)\})$$

Grafos No Dirigidos

Muchas veces nos interesan que los arcos no tengan dirección:



En este caso se llaman grafos no-dirigidos. Formalmente, son relaciones simétricas:

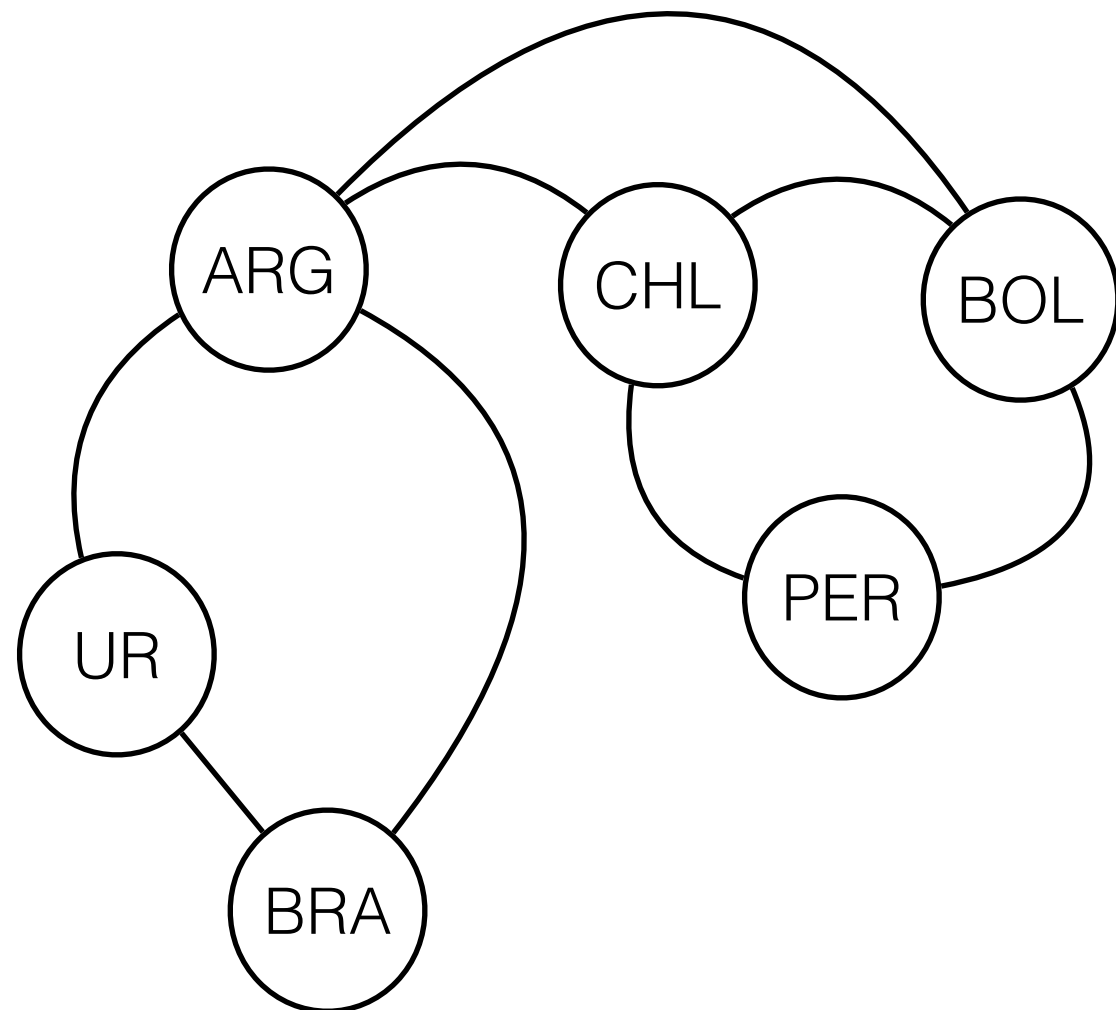
$$(\{v, w, y\}, \{(v, w), (w, y), (y, v), (w, v), (v, y), (v, w)\})$$

Aplicaciones

La mayor parte de los problemas pueden modelarse con grafos.

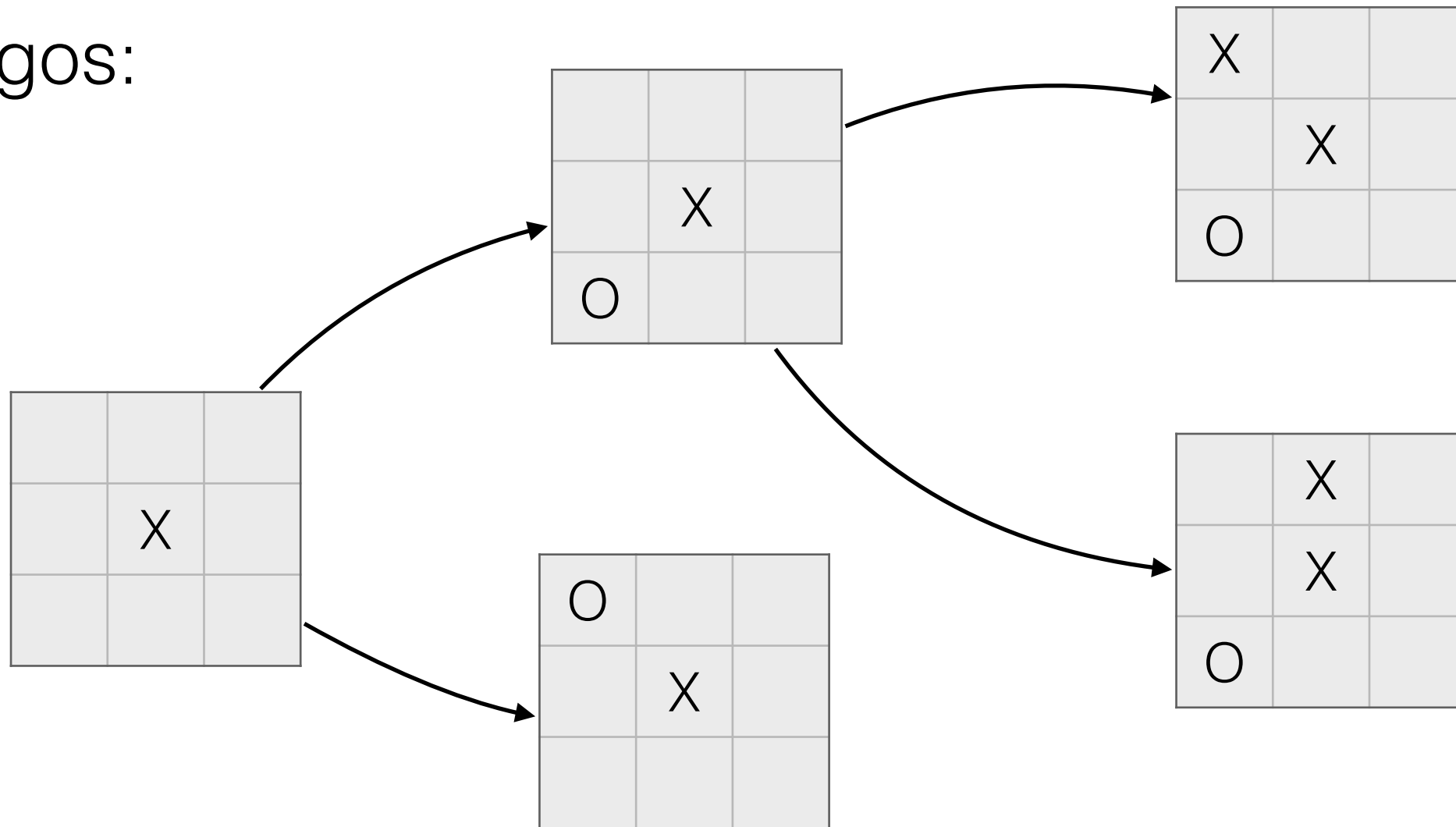


MAPAS



Aplicaciones

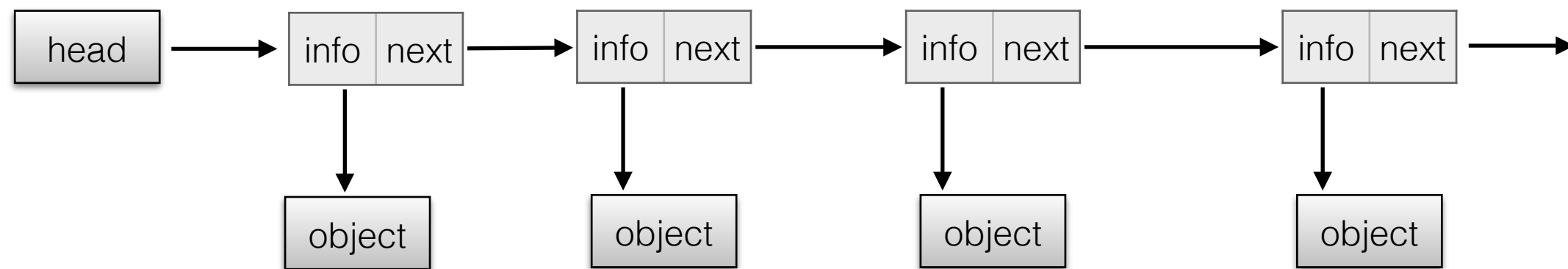
Juegos:



En general el estado de configuraciones de cualquier juego se puede ver como un grafo

Aplicaciones

Representación de heaps (memoria de JAVA)



Esto permite hacer
varios análisis
sobre la memoria

Aplicaciones

- Y muchas más...

application	item	connection
<i>map</i>	intersection	road
<i>web content</i>	page	link
<i>circuit</i>	device	wire
<i>schedule</i>	job	constraint
<i>commerce</i>	customer	transaction
<i>matching</i>	student	application
<i>computer network</i>	site	connection
<i>software</i>	method	call
<i>social network</i>	person	friendship

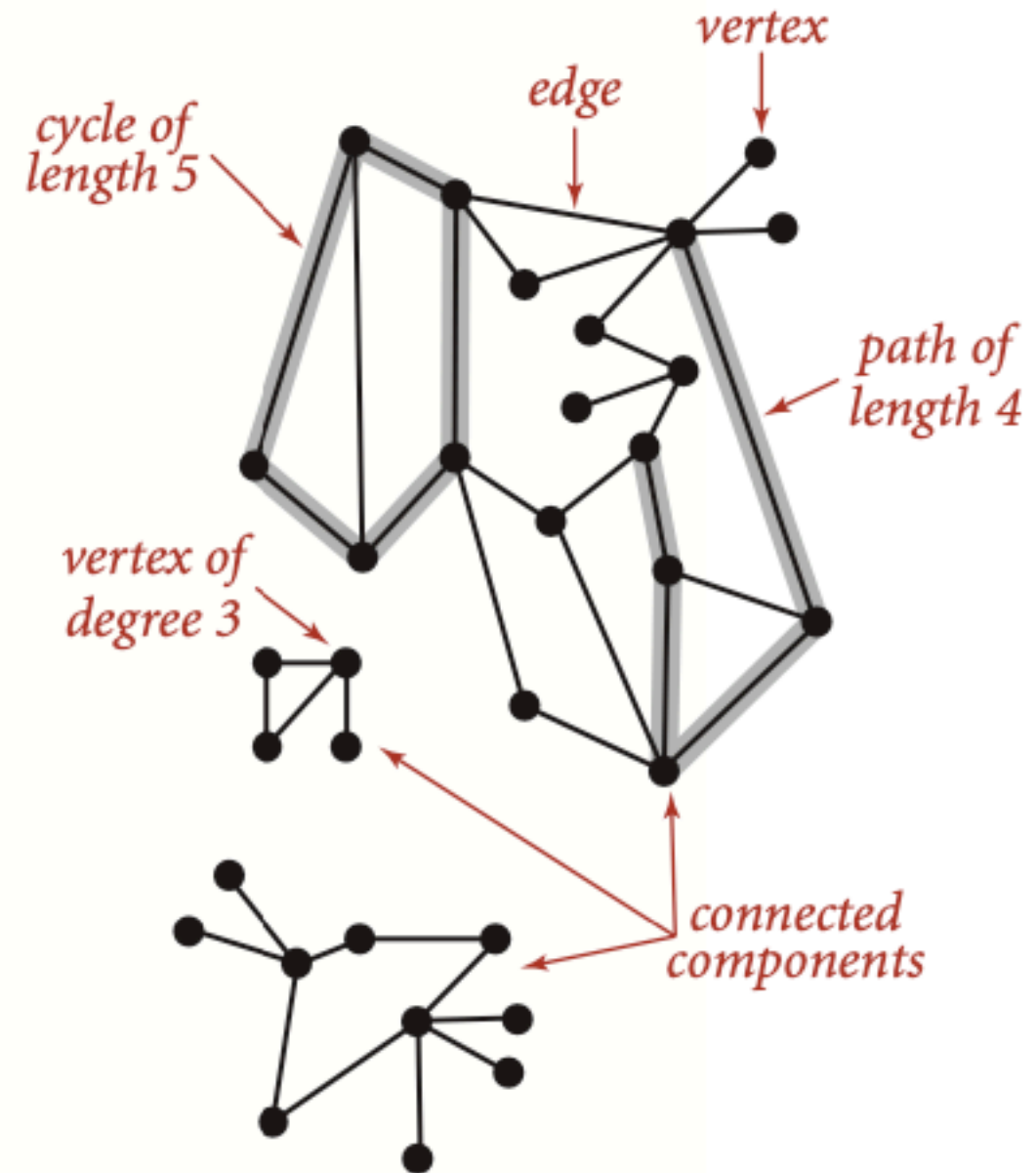
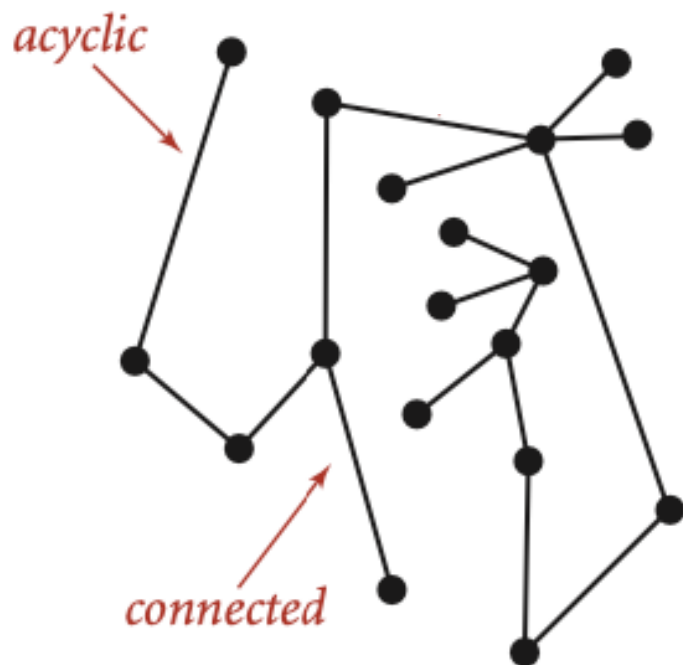
Typical graph applications

Definiciones...

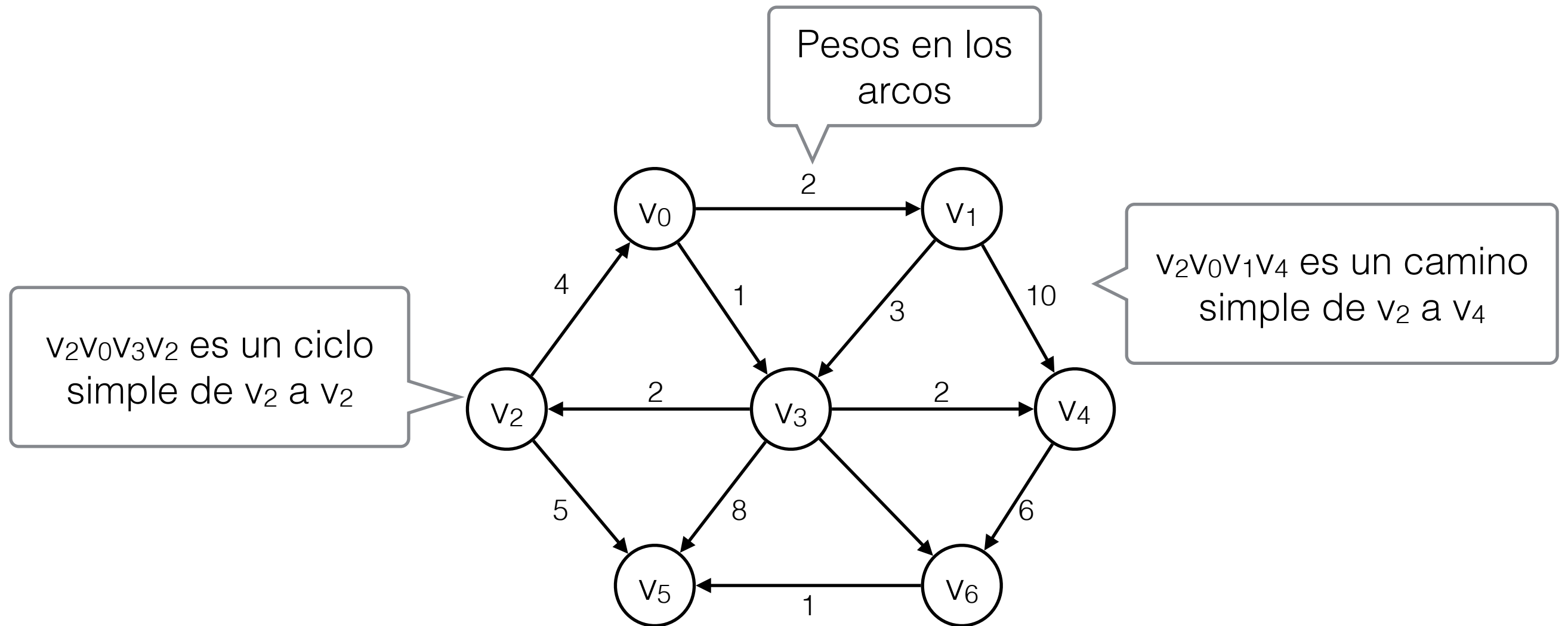
- Un grafo se dice **completo** si cada par de nodos está conectado por un arco
- Un **camino** de un nodo v a un nodo w en un grafo G es una secuencia de nodos adyacentes que comienza en v y termina en w
 - La **longitud** del camino es la cantidad de aristas en el camino
- Si todos los vértices en el camino son distintos el camino se denomina **camino simple**
- Un **ciclo** es un camino de un nodo w a si mismo
- Un **ciclo simple** es un ciclo en el cual no se repiten arcos ni vértices (salvo el primero y el último)
- Un grafo se llama **conexo** si existe un camino desde v a w , para cada par de nodos v y w
- Un grafo con **pesos** es un grafo en el cual los arcos poseen pesos y costos

Definiciones...

- Un grafo se dice **conexo** si hay un camino desde cualquier nodo del grafo a cualquier otro
- Un grafo **no conexo** consiste de un conjunto de **componentes conexas**, que son subgrafos conexos maximales
- El **grado** de un vértice es la cantidad de aristas que entran al vértice
- Un grafo es **acíclico** si no tiene ciclos
- Un **árbol** es un grafo acíclico y conexo



Un Ejemplo



Este es un grafo dirigido (**digrafo**), en muchas aplicaciones trabajamos con grafos dirigidos

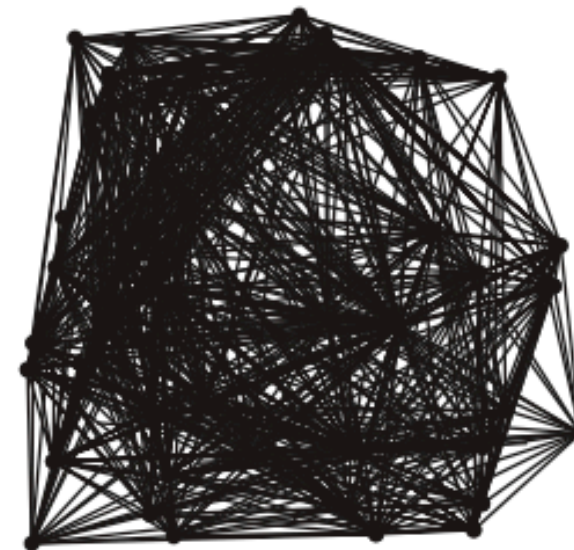
Densidad de grafos

- La densidad es la proporción de aristas que posee un grafo respecto de la cantidad de vértices ($|V|$)
- En un grafo denso la cantidad de aristas es cercano al máximo número posible de aristas
- Un grafo disperso tiene relativamente pocas aristas, típicamente, proporcional a una constante pequeña multiplicada por $|V|$
- En la práctica, en la mayoría de las aplicaciones trabajamos con grafos dispersos

sparse ($E = 200$)



dense ($E = 1000$)

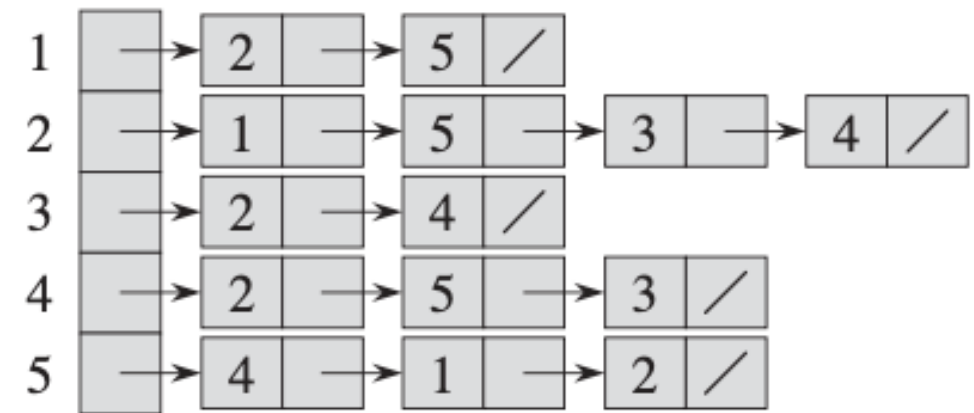
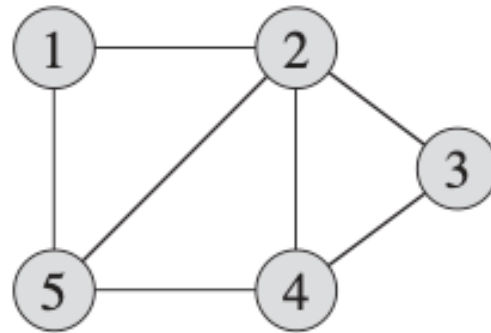


Two graphs ($V = 50$)

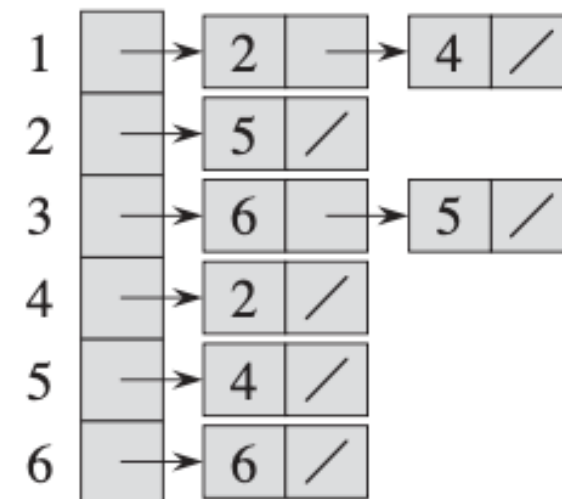
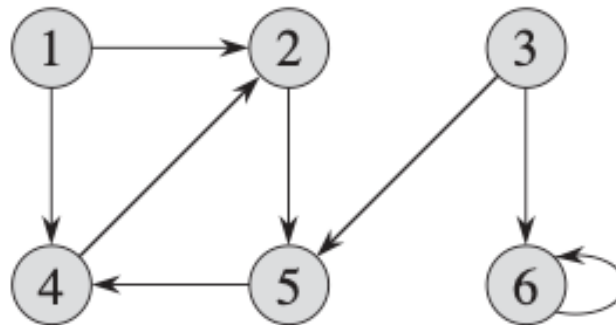
Representaciones de grafos

- Lista de adyacencias: Se crea un arreglo G de N lugares, en donde $G[i]$ contiene una lista con todos los vértices adyacentes al vértice i

- Grafo no dirigido: dos entradas por cada arista que conecta i con j : $i \rightarrow j$ y $j \rightarrow i$



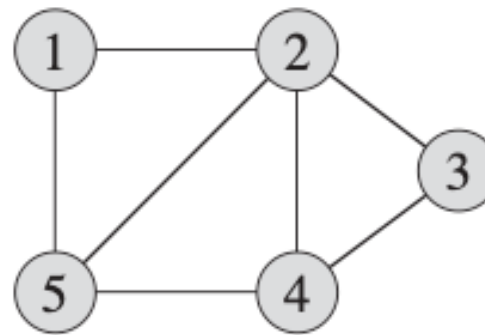
- Grafo dirigido:



Representaciones de grafos

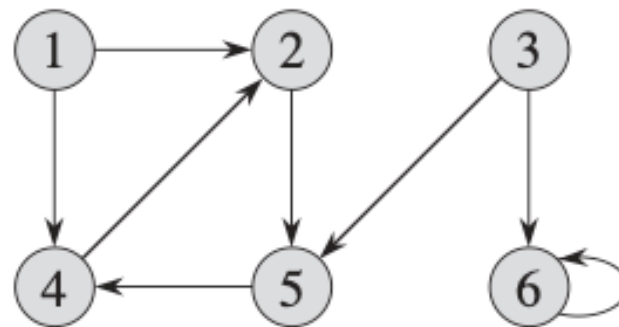
- Matriz de adyacencias: Para un grafo de N vértices se crea una matriz G de $N \times N$, en donde $G[i][j]=1$ si el nodo i está conectado con el nodo j , y $G[i][j]=0$ en otro caso

- Grafo no dirigido: dos 1's por cada arista que conecta i con j :
 $G[i][j]=1$ y $G[j][i]=1$



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- Grafo dirigido:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Grafo no dirigido: Operaciones

```
/**
 * IntGraph represents an undirected graph, where vertices are labeled
 * with integers.
 * Formally, a graph  $G=\langle V,E \rangle$  consists of a set of vertices  $V$ ,
 * and a relation  $E$  in  $V \times V$  that describes the edges of the graph.
 */
public interface IntGraph {
    /**
     * @post Returns the number of vertices in this graph. */
    public int V();
    /**
     * @post Returns the number of edges in this graph. */
    public int E();
    /**
     * @pre  $0 \leq v < V \ \&\& \ 0 \leq w < V$ 
     * @post Adds the undirected edge  $v-w$  to this graph. */
    public void addEdge(int v, int w);
    /**
     * @pre  $0 \leq v < V$ 
     * @post Returns the list of vertices adjacent to vertex  $v$ .*/
    public List<Integer> adj(int v);
}
```

Código fuente basado en: <https://github.com/kevin-wayne/algs4>

Grafo no dirigido: Operaciones

```
/**
 * IntGraph represents an undirected graph, where vertices are labeled
 * with integers.
 * Formally, a graph  $G=\langle V,E \rangle$  consists of a set of vertices  $V$ ,
 * and a relation  $E$  in  $V \times V$  that describes the edges of the graph.
 */
public interface IntGraph {
    /**
     * @post Returns the number of vertices in this graph. */
    public int V();
    /**
     * @post Returns the number of edges in this graph. */
    public int E();
    /**
     * @pre  $0 \leq v < V$  &  $0 \leq w < V$ 
     * @post Adds the undirected edge  $v-w$  to this graph. */
    public void addEdge(int v, int w);
}
```

Para facilitar la comprensión de los algoritmos, vamos a considerar primero grafos en donde los nodos sólo almacenan enteros, y luego daremos una implementación más general

Grafo no dirigido: Algunas consideraciones

- Podríamos añadir fácilmente operaciones para agregar vértices, eliminar vértices, eliminar una arista, chequear si el grafo contiene una arista dada, etc..
 - Por ejemplo, podríamos usar conjuntos o hashes en lugar de listas de adyacencia para obtener una mayor eficiencia para consultar la existencia de una arista
- No vamos a hacerlo por varias razones:
 - Los problemas que resolveremos no las necesitan (muchos problemas sobre grafos no requieren de estas operaciones)
 - En los casos en que se requieren estas operaciones, o se invocan relativamente poco, o para listas de adyacencia cortas (grafos dispersos), por lo que la penalidad de usar listas no es alta
 - Usando listas de adyacencia las implementaciones se simplifican, y podemos enfocarnos mejor en entender los algoritmos sobre grafos

Grafo no dirigido: Implementación

```
public class AdjacencyListIntGraph implements IntGraph {
    // Number of vertices in the graph
    private final int V;
    // Number of edges in the graph
    private int E;
    // Adjacency lists
    private List<Integer>[] adj;

    /**
     * @pre V >= 0
     * @post Initializes a graph with V vertices and 0 edges
     */
    public AdjacencyListIntGraph(int V) {
        if (V < 0)
            throw new IllegalArgumentException("Number of vertices must be non-
negative");
        this.V = V;
        this.E = 0;
        adj = new LinkedList[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedList<Integer>();
        }
    }
}
```

Grafo no dirigido: Implementación

```
/**
 * @pre 0 <= v < V && 0 <= w < V
 * @post Adds the undirected edge v-w to this graph.
 */
public void addEdge(int v, int w) {
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v +
            " is not between 0 and " + (V-1));
    if (w < 0 || w >= V)
        throw new IllegalArgumentException("vertex " + w +
            " is not between 0 and " + (V-1));
    E++;
    adj[v].add(w);
    adj[w].add(v);
}
```

Grafo no dirigido: Implementación

```
/**
 * @pre 0 <= v < V
 * @post Returns the list of vertices adjacent to vertex v.
 */
public List<Integer> adj(int v) {
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v +
            " is not between 0 and " + (V-1));
    return adj[v];
}

/**
 * @post Returns the number of vertices in this graph.
 */
public int V() {
    return V;
}

/**
 * @post Returns the number of edges in this graph.
 */
public int E() {
    return E;
}
```

Grafo no dirigido: Implementación

```
/**
 * @pre 0 <= v < V
 * @post Returns the list of vertices adjacent to vertex v.
 */
public List<Integer> adj(int v) {
    if (v < 0 || v >= V)
        throw new IllegalArgumentException("vertex " + v +
            " is not between 0 and " + (V-1));
    return adj[v];
}

/**
 * @post Returns the number of vertices in this graph.
 */
public int V() {
    return V;
}

/**
```

Ejercicio: Implementar grafos no dirigidos con matrices de adyacencia

```
}
```

Eficiencia de las implementaciones de grafos

underlying data structure	space	add edge $v-w$	check whether w is adjacent to v	iterate through vertices adjacent to v
<i>list of edges</i>	E	1	E	E
<i>adjacency matrix</i>	V^2	1	1	V
<i>adjacency lists</i>	$E + V$	1	$\text{degree}(v)$	$\text{degree}(v)$
<i>adjacency sets</i>	$E + V$	$\log V$	$\log V$	$\text{degree}(v)$

Order-of-growth performance for typical Graph implementations

- Las matrices de adyacencia son prohibitivas para grafos grandes y dispersos, ya que usan una cantidad cuadrática de espacio
 - E iterar sobre los adyacentes es lineal, lo que también es malo para grafos dispersos
- Las aplicaciones típicas procesan grafos dispersos muy grandes, por lo que la representación con listas de adyacencia suele ser la más usada
 - O reemplazar las listas con conjuntos o hashes si la aplicación lo amerita

Eficiencia de las implementaciones de grafos

underlying data structure	space	add edge $v-w$	check whether w is adjacent to v	iterate through vertices adjacent to v
<i>list of edges</i>	E	1	E	$O(V)$ en el peor caso
<i>adjacency matrix</i>	V^2	1	1	
<i>adjacency lists</i>	$E + V$	1	$\text{degree}(v)$	$\text{degree}(v)$
<i>adjacency sets</i>	$E + V$	$\log V$	$\log V$	$\text{degree}(v)$

Order-of-growth performance for typical Graph implementations

- Las matrices de adyacencia son prohibitivas para grafos grandes y dispersos, ya que usan una cantidad cuadrática de espacio
 - E iterar sobre los adyacentes es lineal, lo que también es malo para grafos dispersos
- Las aplicaciones típicas procesan grafos dispersos muy grandes, por lo que la representación con listas de adyacencia suele ser la más usada
 - O reemplazar las listas con conjuntos o hashes si la aplicación lo amerita

Grafo dirigido: Operaciones

```
/**
 * IntDigraph represents an undirected graph, where vertices are labeled
 * with integers.
 * Formally, a graph  $G=\langle V,E \rangle$  consists of a set of vertices  $V$ ,
 * and a relation  $E$  in  $V \times V$  that describes the edges of the graph.
 */
public interface IntDigraph {
    /**
     * @post Returns the number of vertices in this graph. */
    public int V();
    /**
     * @post Returns the number of edges in this graph. */
    public int E();
    /**
     * @pre  $0 \leq v < V \ \&\& \ 0 \leq w < V$ 
     * @post Adds the directed edge  $v \rightarrow w$  to this graph. */
    public void addEdge(int v, int w);
    /**
     * @pre  $0 \leq v < V$ 
     * @post Returns the list of vertices adjacent to vertex  $v$ .*/
    public List<Integer> adj(int v);
}
```


Grafo dirigido: Implementación

- Para implementar grafos dirigidos con listas de adyacencia sólo tenemos que modificar el método addEdge para que agregue aristas dirigidas

```
public class AdjacencyListIntDigraph implements IntDigraph {  
  
    /** ...código omitido... */  
  
    /**  
     * @pre 0 <= v < V && 0 <= w < V  
     * @post Adds the directed edge v->w to this graph.  
     */  
    public void addEdge(int v, int w) {  
        if (v < 0 || v >= V)  
            throw new IllegalArgumentException("vertex " + v +  
                " is not between 0 and " + (V-1));  
        if (w < 0 || w >= V)  
            throw new IllegalArgumentException("vertex " + w +  
                " is not between 0 and " + (V-1));  
        E++;  
        adj[v].add(w);  
    }  
}
```

Grafo dirigido: Implementación

- Para implementar grafos dirigidos con listas de adyacencia sólo tenemos que modificar el método addEdge para que agregue aristas dirigidas

```
public class AdjacencyListIntDigraph implements IntDigraph {  
  
    /** ...código omitido... **/  
  
    /**  
     * @pre 0 <= v < V && 0 <= w < V  
     * @post Adds the directed edge v->w to this graph.  
     */  
    public void addEdge(int v, int w) {  
        if (v < 0 || v >= V)  
            throw new IllegalArgumentException("vertex " + v +  
                " is not between 0 and " + (V-1));  
        if (w < 0 || w >= V)
```

Ejercicio: Implementar grafos dirigidos con matrices de adyacencia

```
}
```

Recorridos de Grafos

Dos formas básicas de recorrer grafos, que permiten implementar muchos algoritmos sobre estos:

- **Depth-First Search (DFS):** o primero en profundidad, se recorre el grafo tal que siempre nos metemos por un camino hasta completarlo antes de recorrer los otros.
- **Breadth-First Search (BFS):** o primero en amplitud, dado un nodo s se recorren los nodos a distancia 1 de s , luego los nodos a distancia 2 de s , luego los nodos a distancia 3, etc...

Depth-first Search

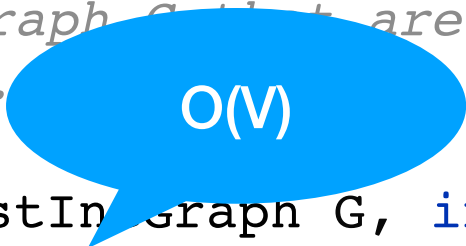
```
/**
 * @pre 0 <= s < G.V().
 * @post Computes the vertices in graph G that are
 * connected to the source vertex s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    boolean[] marked = new boolean[G.V()];
    dfs(G, s);
}

/**
 * @post Recursively traverses the graph G in Depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

Depth-first Search

```
/**
 * @pre 0 <= s < G.V().
 * @post Computes the vertices in graph G that are
 * connected to the source vertex s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    boolean[] marked = new boolean[G.V()];
    dfs(G, s);
}

/**
 * @post Recursively traverses the graph G in Depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```



$O(V)$

Depth-first Search

```
/**
 * @pre 0 <= s < G.V().
 * @post Computes the vertices in graph G that are
 * connected to the source vertex s.
 */
public DepthFirstSearch(AdjacencyListIn Graph G, int s) {
    boolean[] marked = new boolean[G.V()];
    dfs(G, s);
}

/**
 * @post Recursively computes the vertices in graph G that are
 * connected to the source vertex s in depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIn Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

$O(V)$

$O(E)$: Cada arista se recorre una vez (no se vuelven a recorrer para nodos marcados)

Depth-first Search

DFS es $O(V+E)$

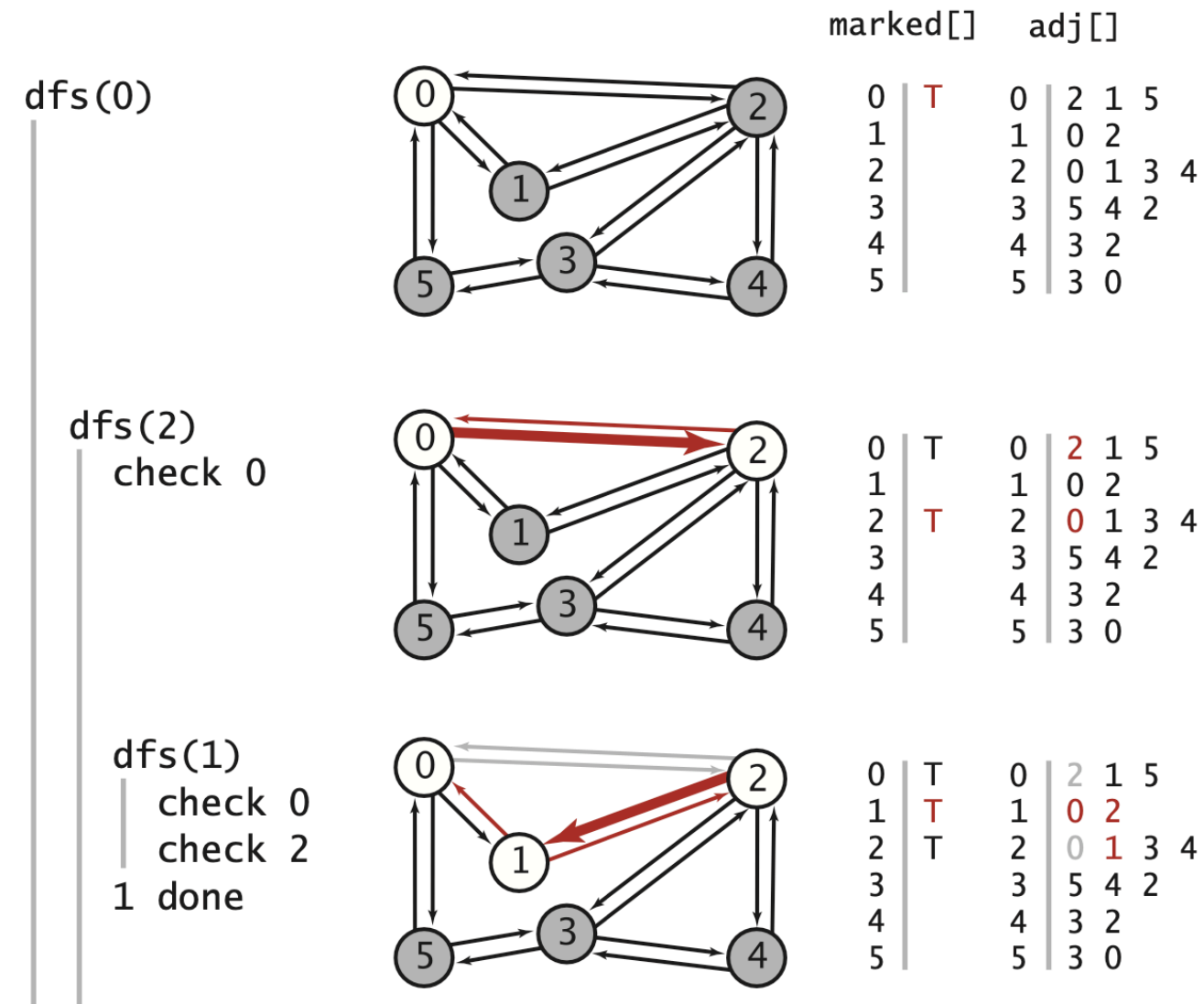
```
/**
 * Returns the vertices in graph G that are
 * reachable from the source vertex s.
 */
public DepthFirstSearch(AdjacencyListIn Graph G, int s) {
    boolean[] marked = new boolean[G.V()];
    dfs(G, s);
}

/**
 * @post Recursively visits all vertices in G in depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIn Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

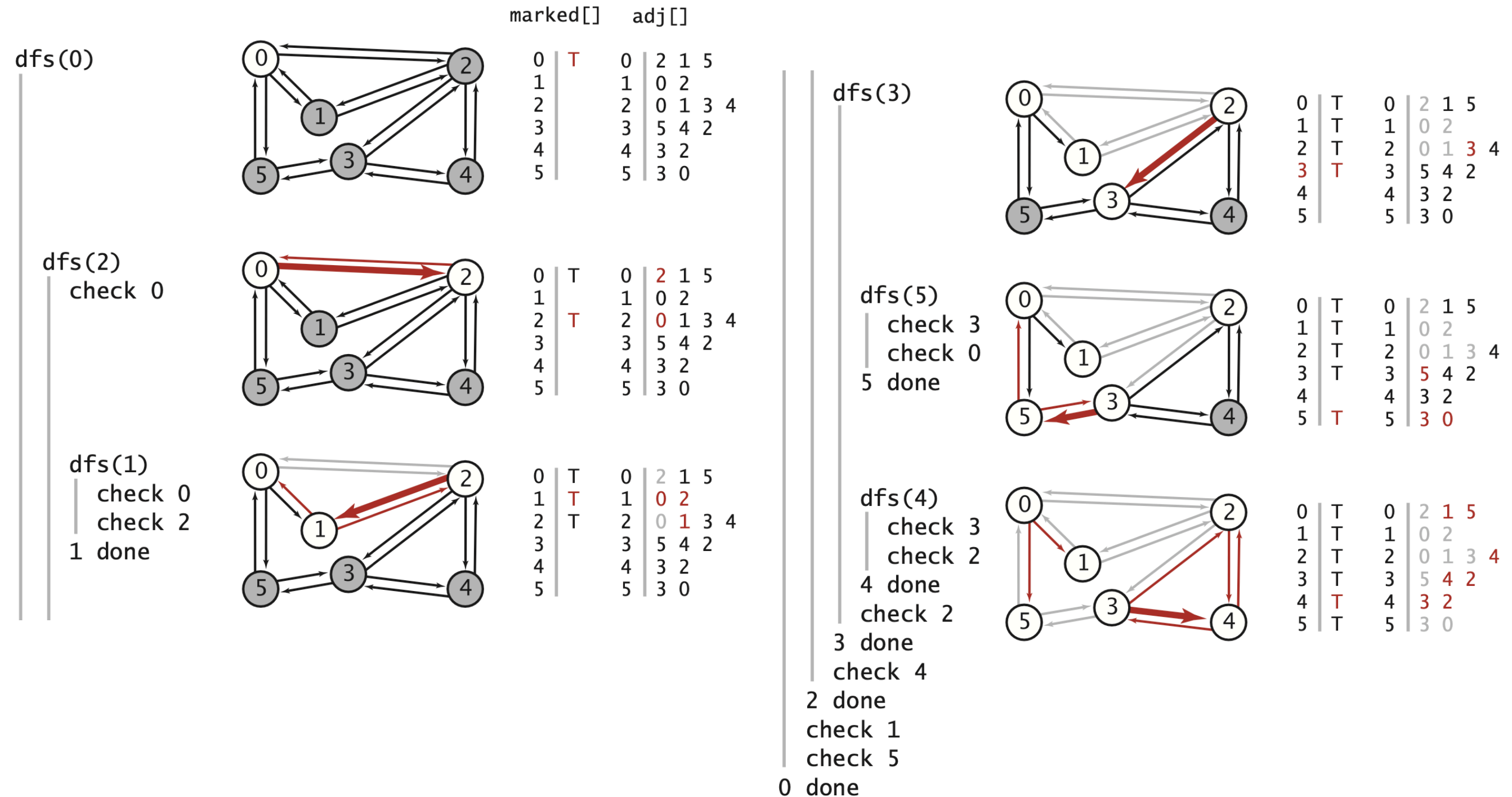
$O(V)$

$O(E)$: Cada arista se
recorre una vez (no se vuelven a
recorrer para nodos
marcados)

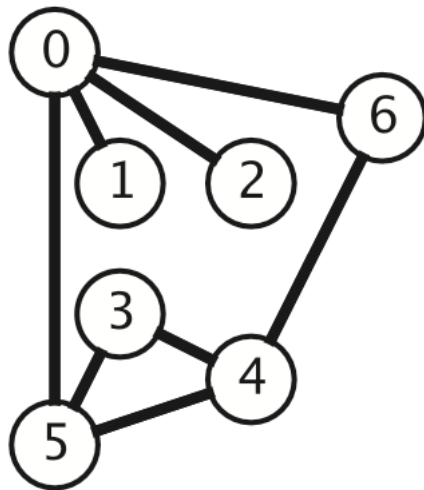
Depth-first Search: Ejemplo



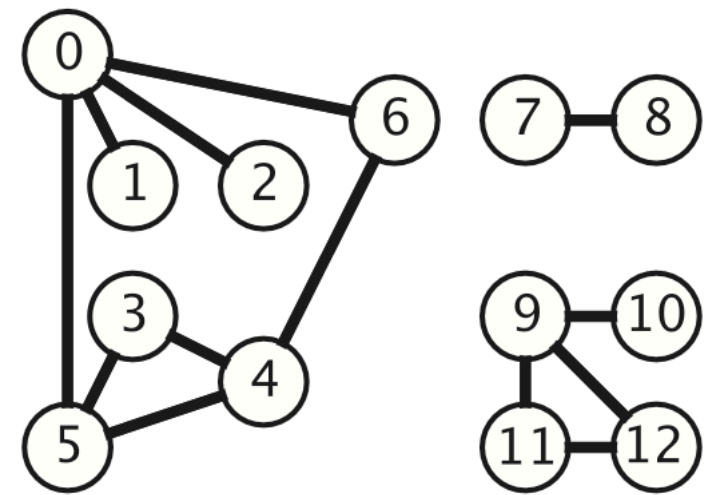
Depth-first Search: Ejemplo



Aplicaciones: Determinar si el grafo es conexo



DFS(G, 0) da count = V ->
el grafo es conexo



DFS(G, 0) da count \neq V ->
el grafo no es conexo

DFS: Determinar si el grafo es conexo

```
/**
 * @pre 0 <= s < G.V().
 * @post Traverses the graph G in Depth-first order starting from s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    boolean[] marked = new boolean[G.V()];
    count = 0;
    dfs(G, s);
}

/**
 * @post Recursively traverses the graph G in Depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    count++;
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

DFS: Determinar si el grafo es conexo

```
/**
 * @pre 0 <= s < G.V().
 * @post Traverses the graph G in Depth-first order starting from s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    boolean[] marked = new boolean[G.V()];
    count = 0;
    dfs(s);
}

// Traverses the graph G in Depth-first order
// starting from s.
// Returns the number of vertices connected to s.
/**
 * @pre 0 <= s < G.V().
 * @post Returns the number of vertices connected to s.
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    count++;
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

Variable global
(atributo de la clase) que al
final de la ejecución tiene la
cantidad de vértices
conectados con s

DFS: Buscar caminos desde un origen

```
/**
 * @pre 0 <= s < G.V().
 * @post Traverses the graph G in Depth-first order starting from s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    this.s = s;
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    this.G = G;
    dfs(G, s);
}

/**
 * @post Recursively traverses the graph G in Depth-first order
 * starting from v.
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

DFS: Buscar caminos desde un origen

```
/**
 * @pre 0 <= s < G.V().
 * @post Traverses the graph G in Depth-first order starting from s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    this.s = s;
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    this.G = G;
    dfs(G, s);
}

/**
 * @post Recursively traverses the graph G in Depth-first order
 *
 * w fue visitado en el camino que viene desde v
 */
public void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

DFS: Buscar caminos desde un origen

```
/**
 * @param s source vertex
 * @param edgeTo array of vertices reached in depth-first order starting from s.
 */
public DepthFirstSearch(AdjacencyListIntGraph G, int s) {
    this.s = s;
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    this.G = G;
    dfs(G, s);
}
```

Podemos armar el camino que conecta s con un vértice v recorriendo edgeTo desde v hasta llegar a s

```
/**
 * @post Recursively traverses the graph G in Depth-first order
 */
private void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

w fue visitado en el camino que viene desde v

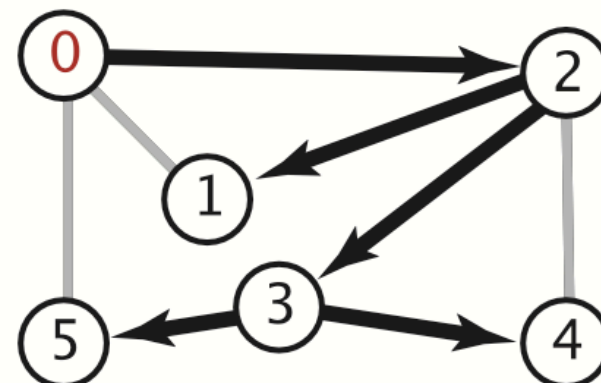
DFS: Buscar caminos desde un origen

Podemos armar el camino que conecta s con un vértice v recorriendo `edgeTo` desde v hasta llegar a s

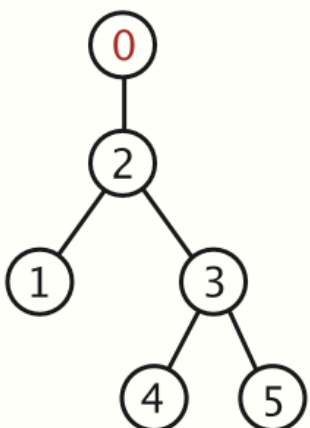
```
/**
 * @param s starting vertex
 * @param G graph
 */
public DepthFirstSearcher(AdjacencyListIntGraph G, int s) {
    this.s = s;
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    this.G = G;
    dfs(G, s);
}
```

```
/**
 * @post Recursively traverses the graph G in Depth-first order
 */
public void dfs(AdjacencyListIntGraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

w fue visitado en el camino que viene desde v



edgeTo[]	
0	
1	2
2	0
3	2
4	3
5	3



DFS: Buscar caminos desde un origen

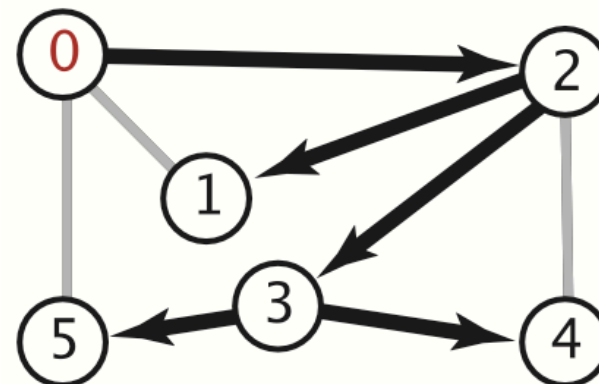
```
/**
 * @pre 0 <= v < V && dfs(G, s) has been executed
 * @post Is there a directed path from the source vertex s to vertex v?
 */
public boolean hasPathTo(int v) {
    isValidVertex(v);
    return marked[v];
}

/**
 * @pre 0 <= v < V && dfs(G, s) has been executed
 * @post Returns a directed path from the source vertex s
 * to vertex v, or null if no such path.
 */
public List<Integer> pathTo(int v) {
    isValidVertex(v);
    if (!hasPathTo(v)) return null;
    LinkedList<Integer> path = new LinkedList<>();
    for (int x = v; x != s; x = edgeTo[x])
        path.addFirst(x);
    path.addFirst(s);
    return path;
}
```

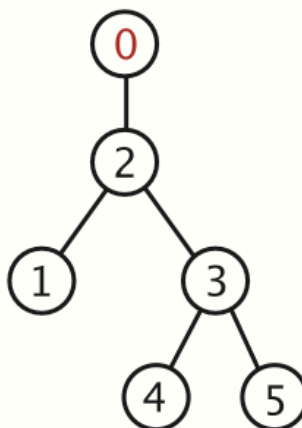
DFS: Buscar caminos desde un origen

```
/**
 * @pre 0 <= v < V && dfs(G, s) has been executed
 * @post Is there a directed path from the source vertex s to vertex v?
 */
public boolean hasPathTo(int v) {
    isValidVertex(v);
    return marked[v];
}

/**
 * @pre 0 <= v < V && dfs(G, s) has been executed
 * @post Returns a directed path from the source vertex s
 * to vertex v, or null if no such path.
 */
public List<Integer> pathTo(int v) {
    isValidVertex(v);
    if (!hasPathTo(v)) return null;
    LinkedList<Integer> path = new LinkedList<>();
    for (int x = v; x != s; x = edgeTo[x])
        path.addFirst(x);
    path.addFirst(s);
    return path;
}
```



edgeTo[]	
0	
1	2
2	0
3	2
4	3
5	3




Breadth-first Search

```
/**
 * @pre  $0 \leq s < G.V()$ .
 * @post Traverses the graph in Breadth-first order starting from  $s$ .
 */
public void bfs(AdjacencyListIntGraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    Queue<Integer> q = new Queue<Integer>();
    marked[s] = true;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                q.enqueue(w);
            }
        }
    }
}
```

Breadth-first Search

```
/**
 * @pre 0 <= s < G.V()
 * @post Traverses the graph in breadth-first order starting from s.
 */
public void bfs(AdjacencyListIntGraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    Queue<Integer> q = new Queue<Integer>();
    marked[s] = true;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                q.enqueue(w);
            }
        }
    }
}
```



Breadth-first Search

```
/**
 * @pre 0 <= s < G.V()
 * @post Traverses the graph in breadth-first order starting from s.
 */
public void bfs(AdjacencyListIntGraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    Queue<Integer> q = new Queue<>();
    marked[s] = true;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                q.enqueue(w);
            }
        }
    }
}
```

$O(V)$

$O(E)$: Cada arista se recorre una vez (no se vuelven a recorrer para nodos marcados)

Breadth-first Search

BFS es $O(V+E)$

$O(V)$

$O(E)$: Cada arista se
recorre una vez (no se vuelven a
recorrer para nodos
marcados)

```
*/
// Returns the breadth-first order starting from s.
public void bfs(AdjacencyListIntGraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    Queue<Integer> q = new Queue<>();
    marked[s] = true;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                q.enqueue(w);
            }
        }
    }
}
```

Breadth-first Search

BFS es $O(V+E)$

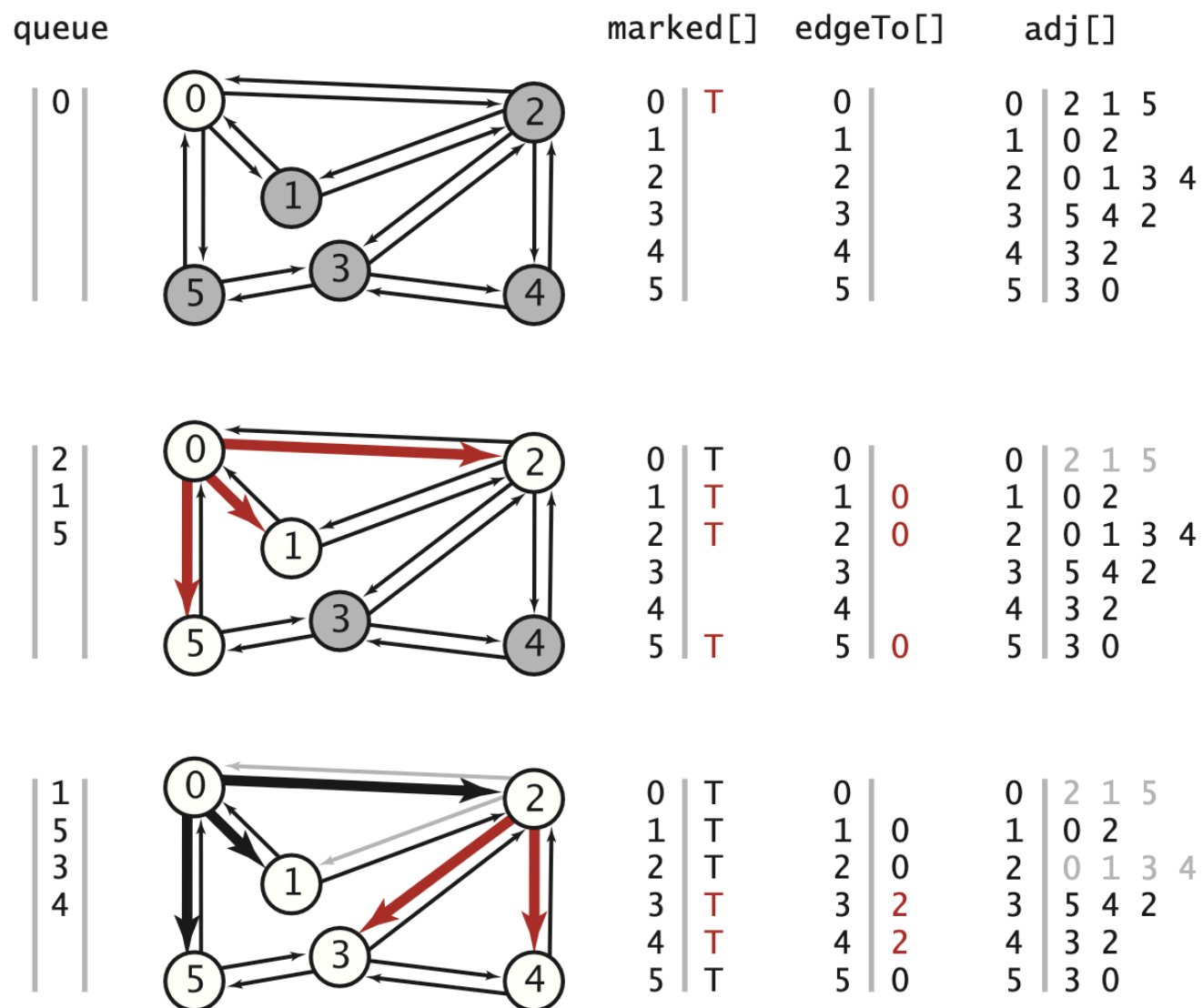
$O(V)$

$O(E)$: Cada arista se
recorre una vez (no se vuelven a
recorrer para nodos
marcados)

```
*/
// Returns the breadth-first order starting from s.
public void bfs(AdjacencyListIntGraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    Queue<Integer> q = new Queue<>();
    marked[s] = true;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
            }
        }
    }
}
```

Ejercicio: Si cambiamos la cola por una pila obtenemos una implementación iterativa de DFS. Implementar DFS iterativo.

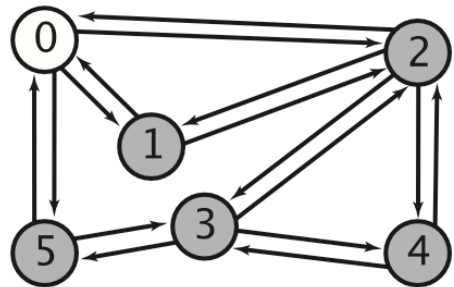
Breadth-first Search: Ejemplo



Breadth-first Search: Ejemplo

queue

0



marked[]

0 T
1
2
3
4
5

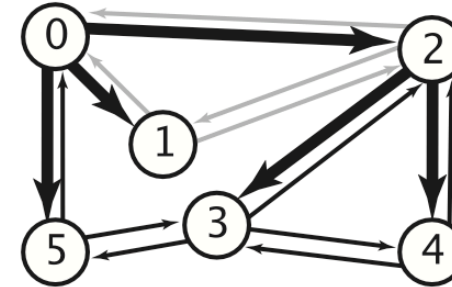
edgeTo[]

0
1
2
3
4
5

adj[]

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

5
3
4

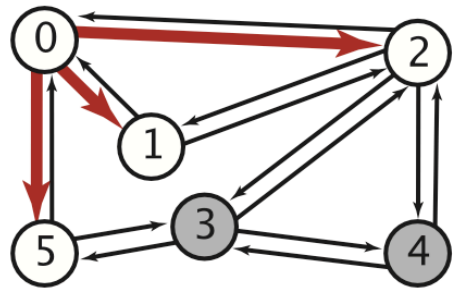


0 T
1 T
2 T
3 T
4 T
5 T

0
1 0
2 0
3 2
4 2
5 0

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

2
1
5

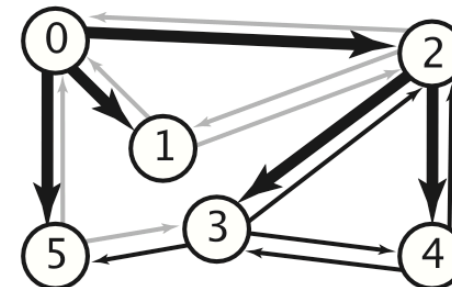


0 T
1 T
2 T
3
4
5 T

0 0
1 0
2 0
3
4
5 0

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

3
4

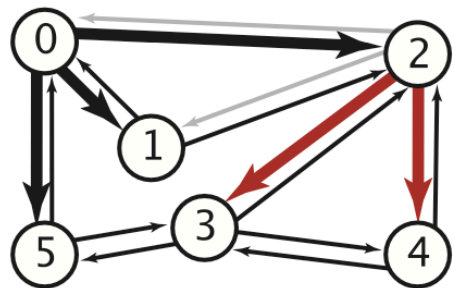


0 T
1 T
2 T
3 T
4 T
5 T

0
1 0
2 0
3 2
4 2
5 0

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

1
5
3
4

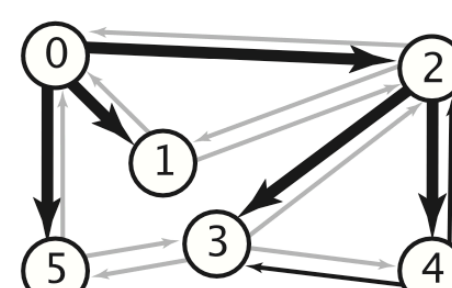


0 T
1 T
2 T
3 T
4 T
5 T

0 0
1 0
2 0
3 2
4 2
5 0

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

4

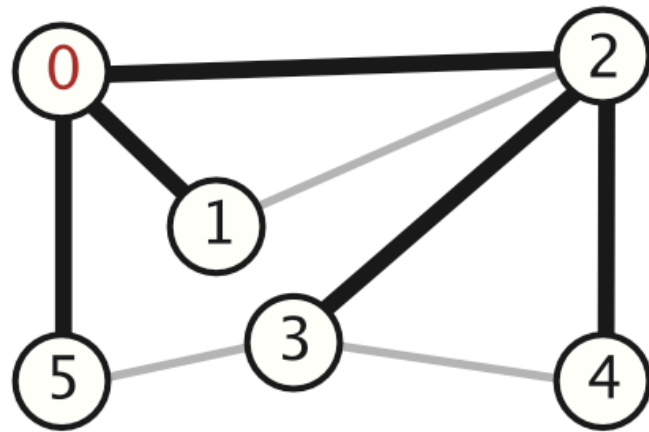


0 T
1 T
2 T
3 T
4 T
5 T

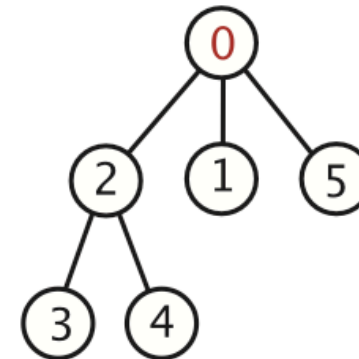
0
1 0
2 0
3 2
4 2
5 0

0 2 1 5
1 0 2
2 0 1 3 4
3 5 4 2
4 3 2
5 3 0

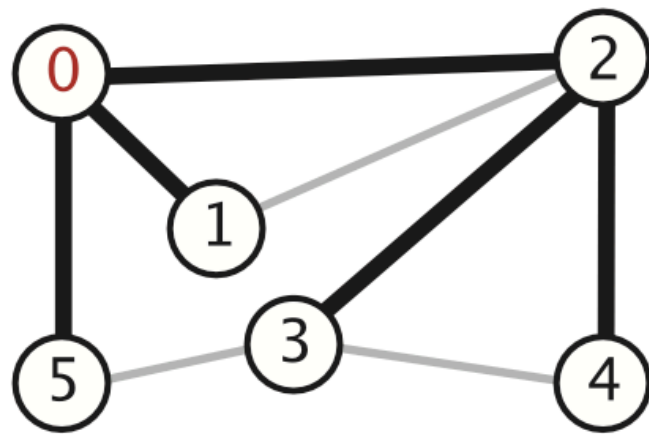
Breadth-first Search: Camminos más cortos



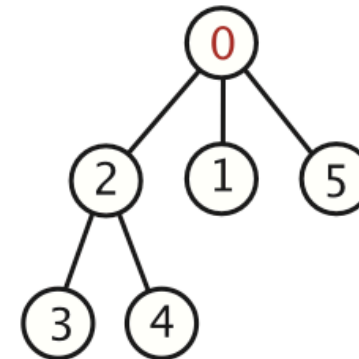
edgeTo[]	
0	
1	0
2	0
3	2
4	2
5	0



Breadth-first Search: Caminos más cortos

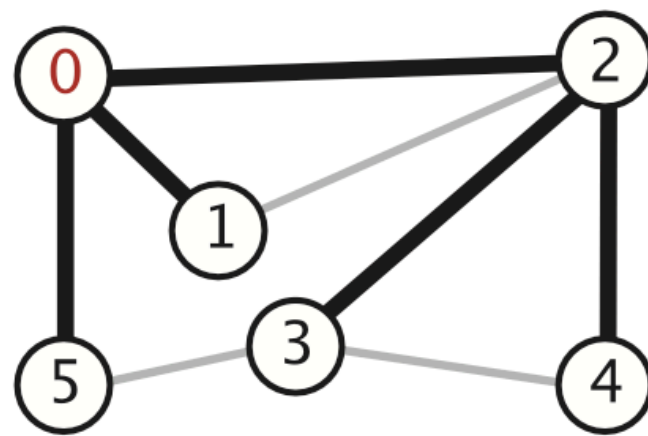


edgeTo[]	
0	
1	0
2	0
3	2
4	2
5	0

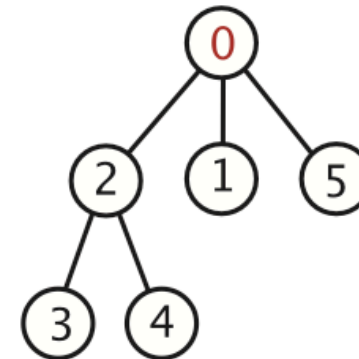


BFS computa los caminos más cortos (con mínima cantidad de aristas)
entre el nodo origen y los nodos que alcanza

Breadth-first Search: Caminos más cortos



edgeTo[]	
0	
1	0
2	0
3	2
4	2
5	0

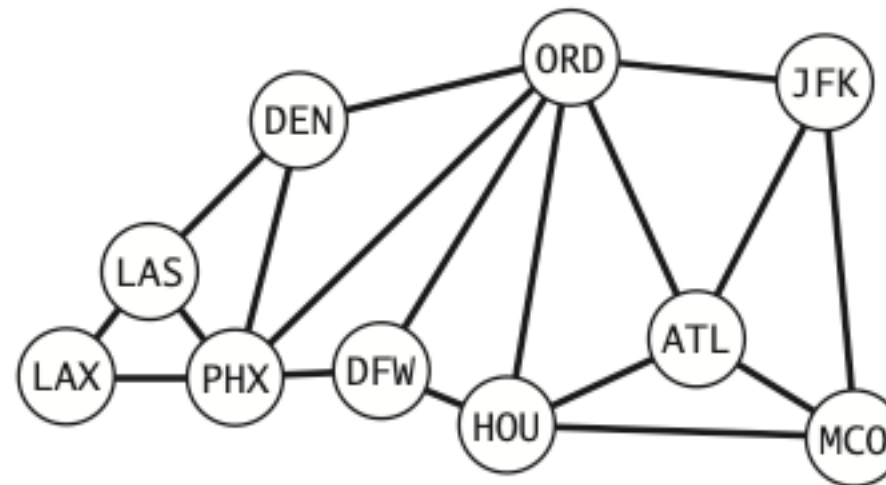


BFS computa los caminos más cortos (con mínima cantidad de aristas) entre el nodo origen y los nodos que alcanza

Primero pone en la cola todos los nodos a distancia 0 de s (s mismo), luego todos los nodos a distancia 1 de s, después los nodos a distancia 2, y así sucesivamente

Grafos genéricos

- Si bien fueron útiles para explicar más fácilmente los algoritmos, los grafos de enteros son muy limitados
- En muchas de las aplicaciones queremos guardar información en los nodos del grafo
 - Por ejemplo, el siguiente grafo modela (algunas de) las rutas aéreas entre aeropuertos de estados unidos

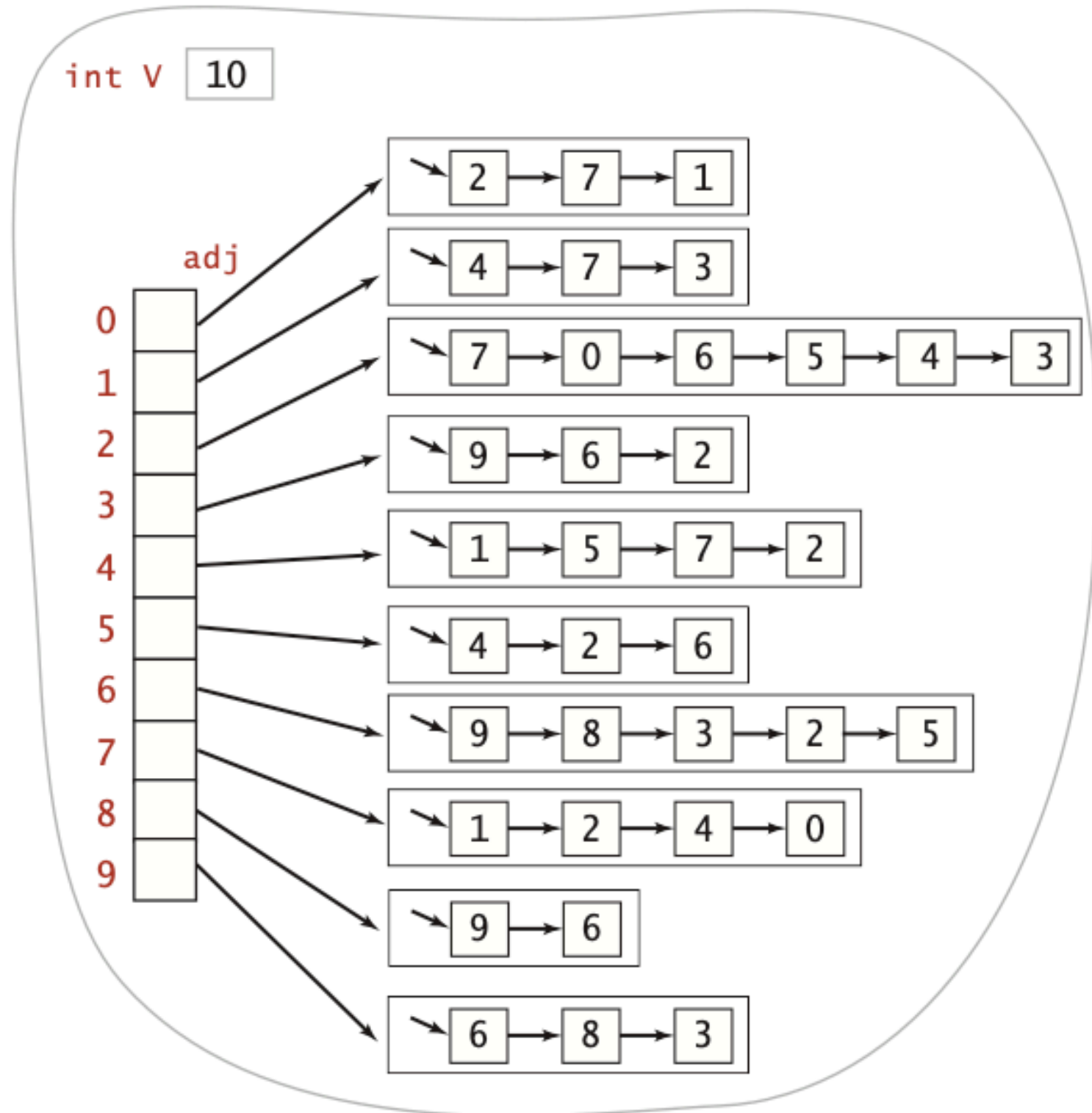
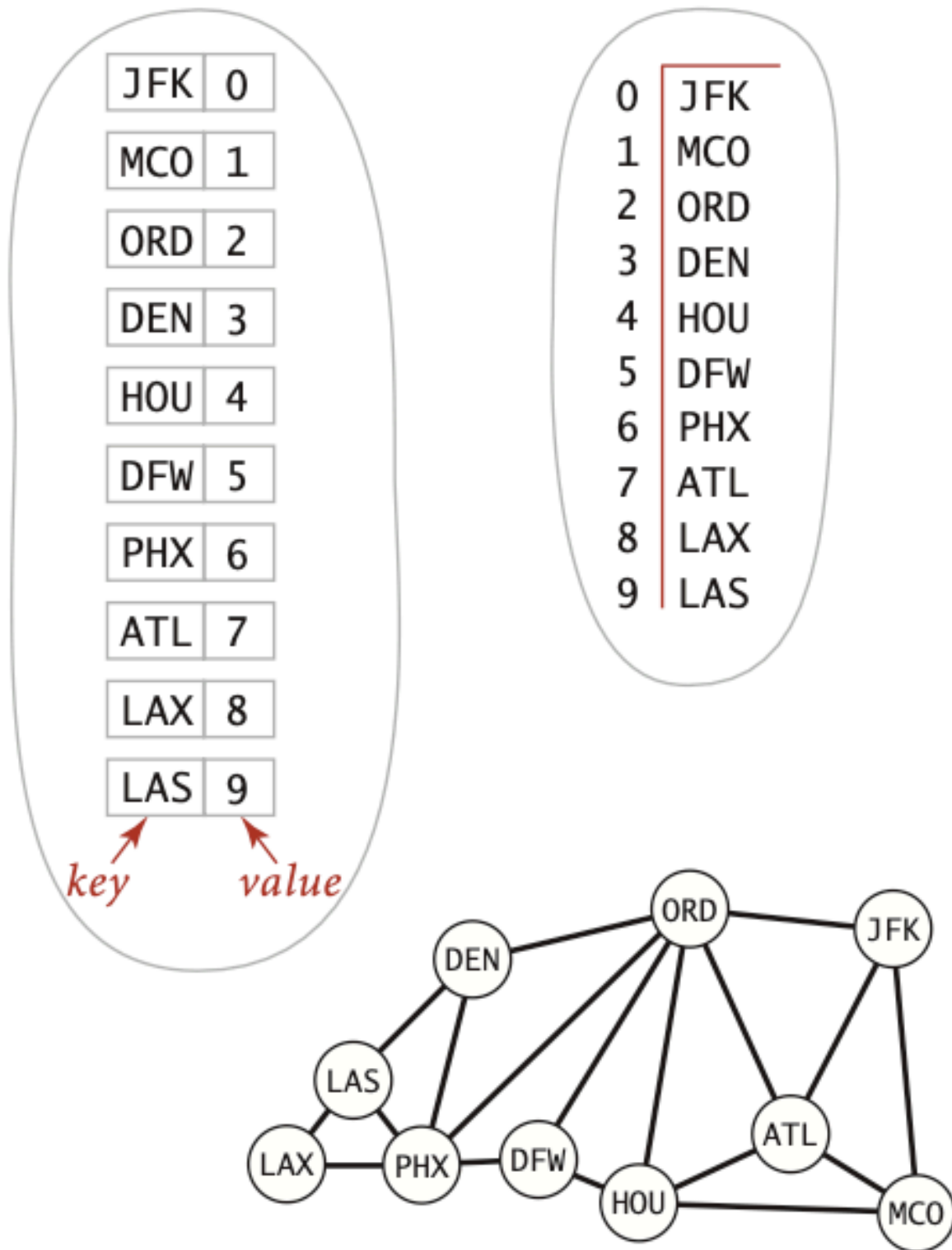


- Podemos extender fácilmente nuestra implementación anterior para soportar este tipo de aplicaciones
- Por ejemplo, asociando la información de los vértices con los índices (enteros) de los nodos en una estructura aparte (map)

Grafos genéricos

```
public interface Graph<T extends Comparable<? super T>> {  
    /**  
     * @post Returns the number of vertices in this graph. */  
    public int V();  
    /**  
     * @post Returns the number of edges in this graph. */  
    public int E();  
    /**  
     * @pre !containsVertex(v).  
     * @post Adds the vertex with label v to this graph. */  
    public void addVertex(T v);  
    /**  
     * @post Returns true iff there is a vertex with label v  
     * in this graph. */  
    public boolean containsVertex(T v);  
    /**  
     * @pre v and w are vertices of the graph  
     * @post Adds the undirected edge v-w to this graph.*/  
    public void addEdge(T v, T w);  
}
```

Grafos genéricos



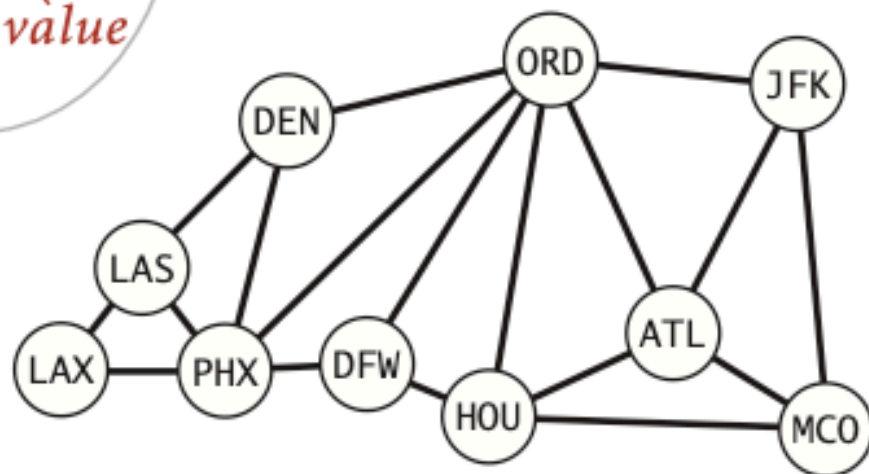
map: String -> Integer

Grafos genéricos

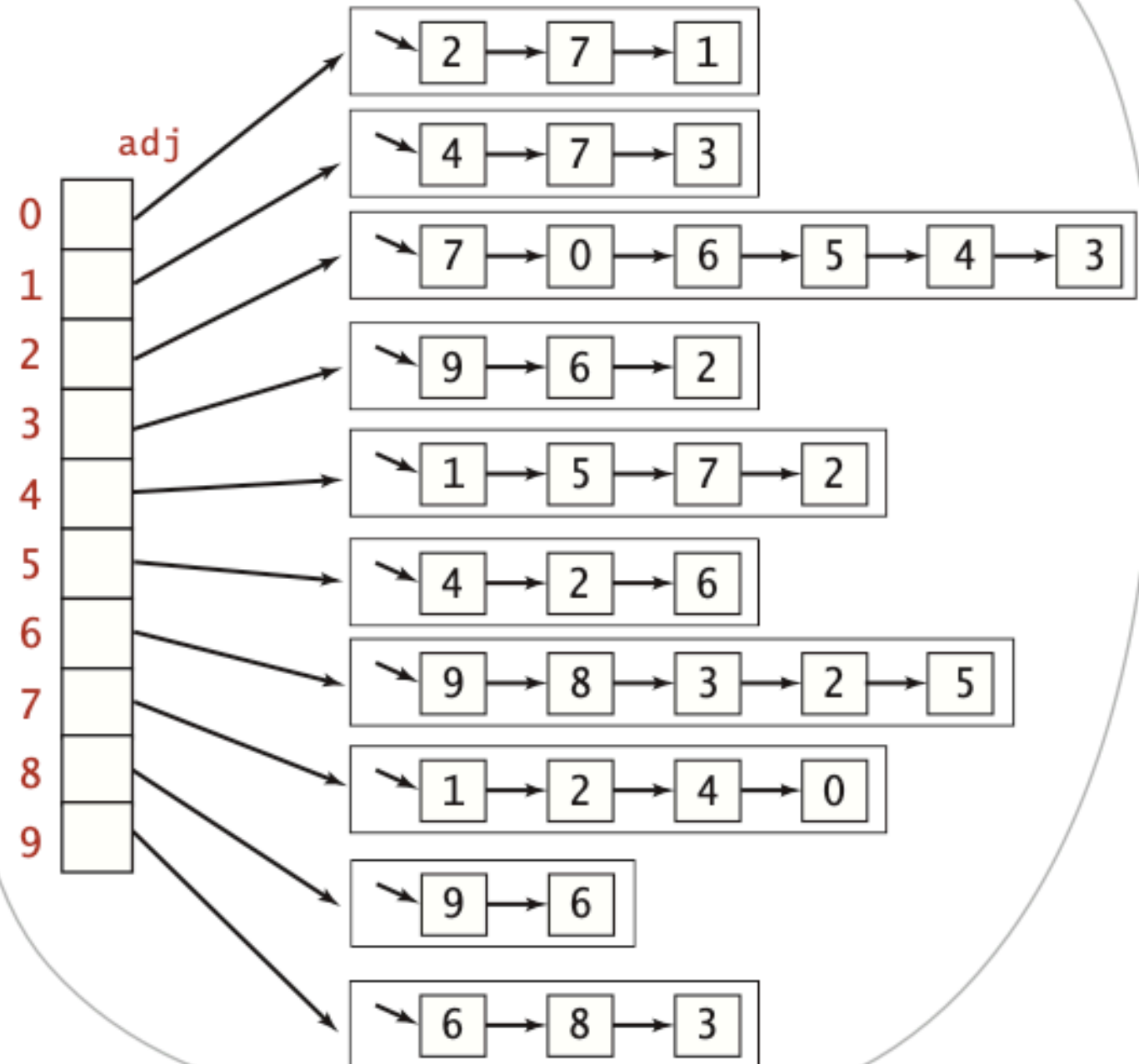
JFK	0
MCO	1
ORD	2
DEN	3
HOU	4
DFW	5
PHX	6
ATL	7
LAX	8
LAS	9

key value

0	JFK
1	MCO
2	ORD
3	DEN
4	HOU
5	DFW
6	PHX
7	ATL
8	LAX
9	LAS



int V 10



Grafos genéricos

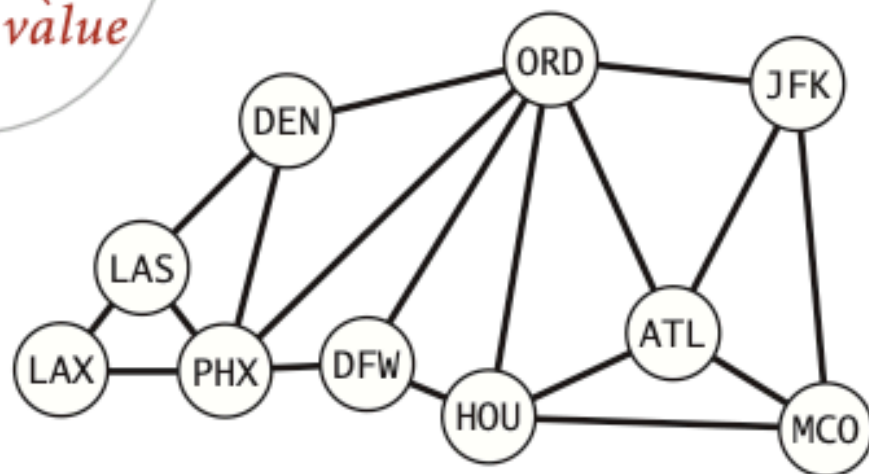
map: String -> Integer

keys: int -> String

JFK	0
MCO	1
ORD	2
DEN	3
HOU	4
DFW	5
PHX	6
ATL	7
LAX	8
LAS	9

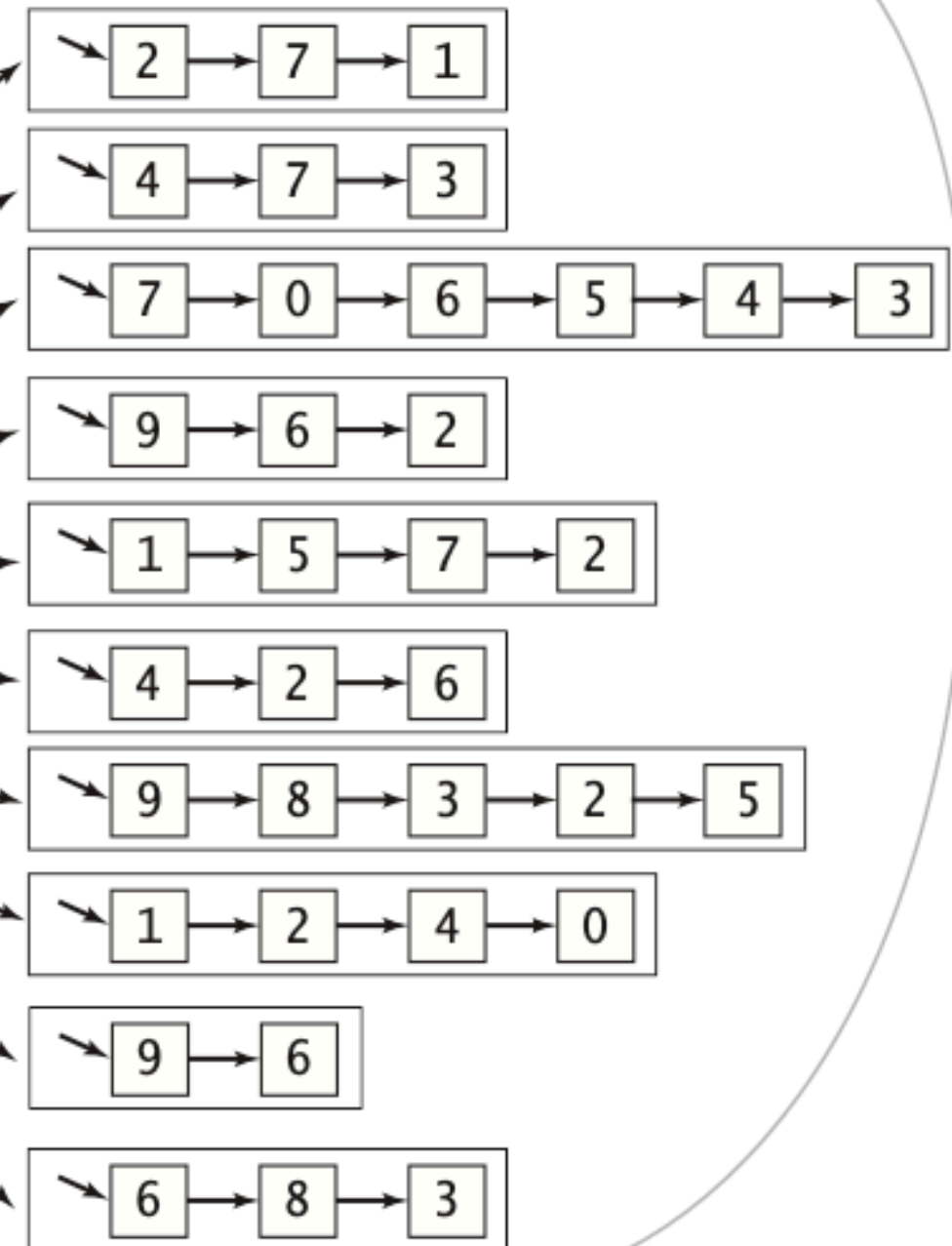
key value

0	JFK
1	MCO
2	ORD
3	DEN
4	HOU
5	DFW
6	PHX
7	ATL
8	LAX
9	LAS



int V 10

adj



Grafos genéricos

map: String -> Integer

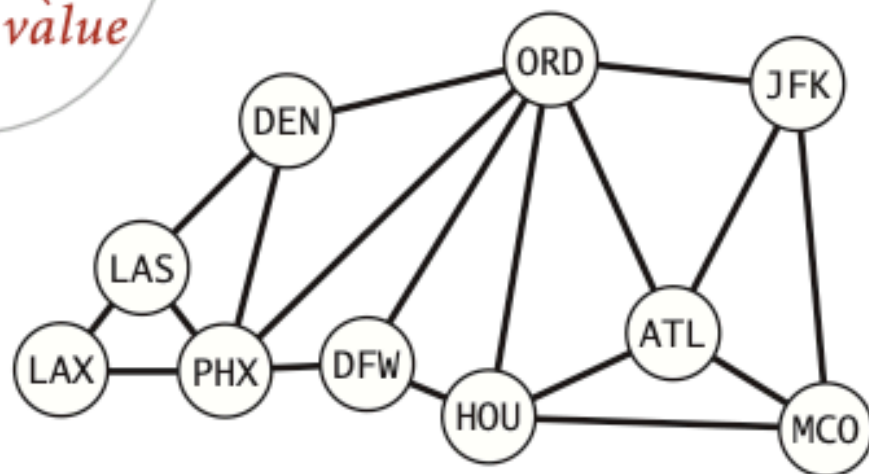
keys: int -> String

adj: int -> List<Integer>

JFK	0
MCO	1
ORD	2
DEN	3
HOU	4
DFW	5
PHX	6
ATL	7
LAX	8
LAS	9

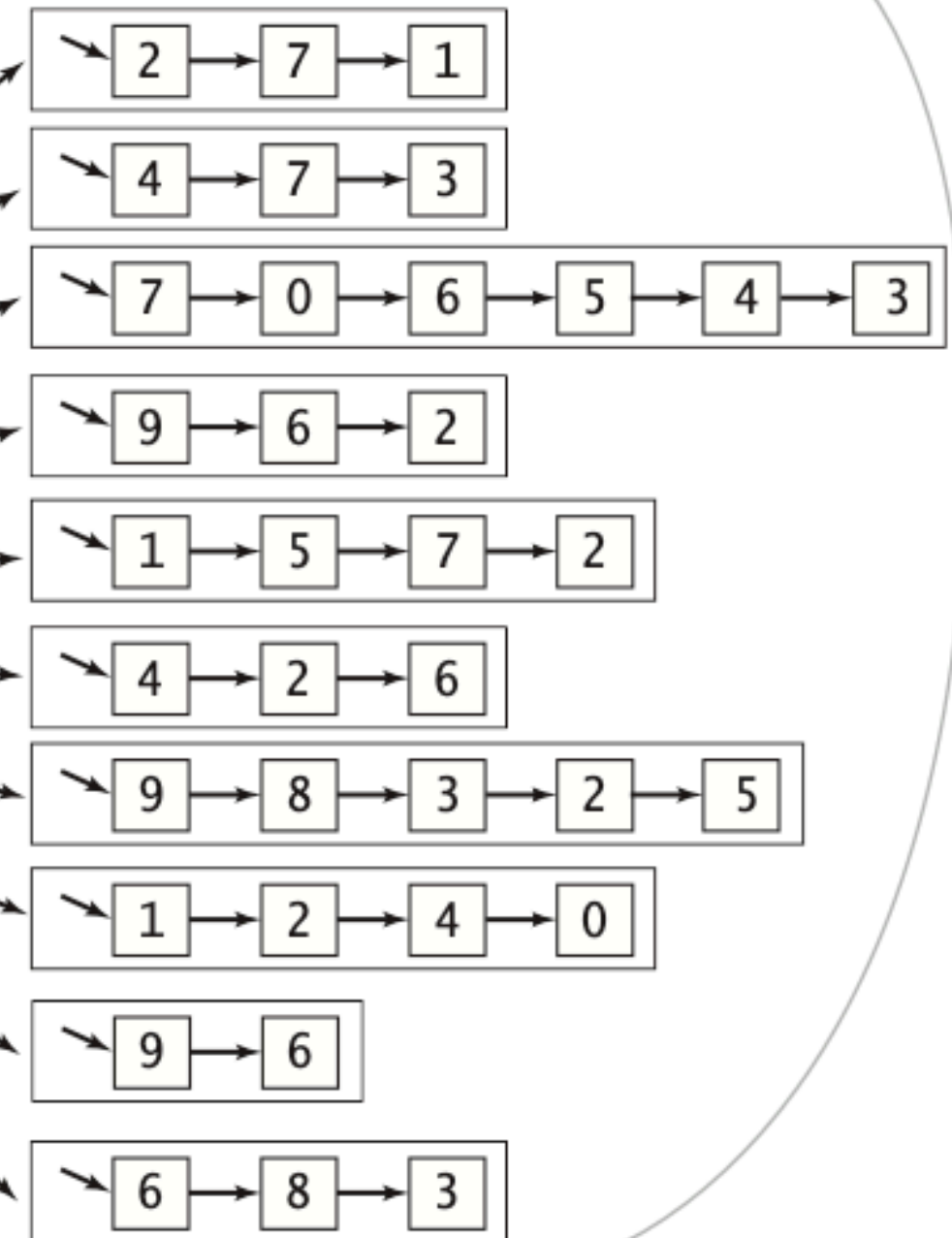
key value

0	JFK
1	MCO
2	ORD
3	DEN
4	HOU
5	DFW
6	PHX
7	ATL
8	LAX
9	LAS



int V 10

adj



Grafos genéricos: Implementación

```
/**
 * AdjacencyListGraph implements a generic undirected graph.
 * Formally, a graph  $G=\langle V,E \rangle$  consists of a set of vertices  $V$ ,
 * and a relation  $E$  in  $V \times V$  that describes the edges of the graph.
 *
 * This implementation uses an adjacency-lists representation, which
 * is a vertex-indexed array of List objects.
 */
public class AdjacencyListGraph<T extends Comparable<? super T>>
    implements Graph<T> {
    // Number of vertices in the graph
    private int V;
    // Number of edges in the graph
    private int E;
    // T -> index
    private TreeMap<T, Integer> map;
    // index -> T
    private T[] keys;
    // Adjacency lists
    private List<Integer>[] adj;
```

Grafos genéricos: Implementación

```
/**
 * @pre V>=0
 * @post Initializes an empty graph that can store up to V vertices */
public AdjacencyListGraph(int V) {
    if (V < 0)
        throw new IllegalArgumentException("Number of vertices must be non-
negative");
    this.V = 0;
    this.E = 0;
    adj = new LinkedList[V];
    map = new TreeMap<>();
    keys = (T[]) new Comparable[V];
}

/**
 * @pre 0<=v<V
 * @post Returns the name of the vertex associated with the integer v. */
T nameOf(int v) {
    return keys[v];
}

/**
 * @pre containsVertex(v).
 * @post Returns the integer associated with the vertex named v. */
int indexOf(T v) {
    return map.get(v);
}
```

Grafos genéricos: Implementación

```
/**
 * @pre V>=0
 * @post Initializes an empty graph that can store up to V vertices */
public AdjacencyListGraph(int V) {
    if (V < 0)
        throw new IllegalArgumentException("V must be non-negative");
    this.V = 0;
    this.E = 0;
    adj = new LinkedList[V];
    map = new TreeMap<>();
    keys = (T[]) new Comparable[V];
}

/**
 * @pre 0<=v<V
 * @post Returns the name of the vertex associated with the integer v. */
T nameOf(int v) {
    return keys[v];
}

/**
 * @pre containsVertex(v).
 * @post Returns the integer associated with the vertex named v. */
int indexOf(T v) {
    return map.get(v);
}
```

Podemos usar un arreglo que vaya creciendo dinámicamente, o un map, si fuera necesario para la aplicación particular

Grafos genéricos: Implementación

```
/**
 * @post Returns true iff there is no vertex with label v
 * in this graph.
 */
public boolean containsVertex(T v) {
    return map.containsKey(v);
}

/**
 * @pre !containsVertex(v).
 * @post Adds the vertex with label v to this graph.
 */
public void addVertex(T v) {
    if (containsVertex(v))
        throw new IllegalArgumentException("Vertex already in the graph");
    int vid = V++;
    map.put(v, vid);
    adj[vid] = new LinkedList<>();
    keys[vid] = v;
}
```

Grafos genéricos: Implementación

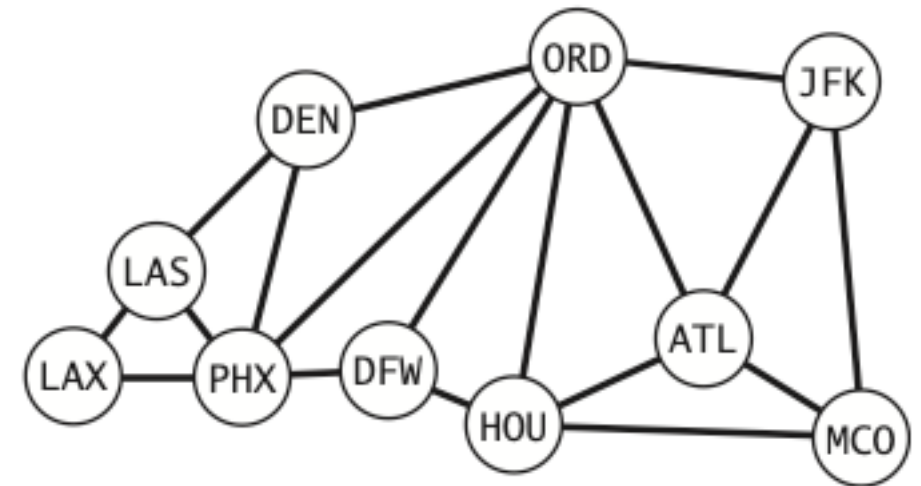
```
/**
 * @pre 0 <= v < V && 0 <= w < V
 * @post Adds the undirected edge v-w to this graph.
 */
public void addEdge(T v, T w) {
    if (!containsVertex(v))
        throw new IllegalArgumentException("vertex " + v +
            " is not between 0 and " + (V-1));
    if (!containsVertex(w))
        throw new IllegalArgumentException("vertex " + w +
            " is not between 0 and " + (V-1));
    E++;
    int vid = indexOf(v);
    int wid = indexOf(w);
    adj[vid].add(wid);
    adj[wid].add(vid);
}
```

Grafos genéricos: Implementación

```
/**
 * @post Returns a string representation
 * of this graph.
 */
public String toString() {
    String s = "";
    for (int v = 0; v < V; v++) {
        s += nameOf(v) + ": ";
        for (int w : adj[v]) {
            s += nameOf(w) + " ";
        }
        s += '\n';
    }
    return s;
}
```


Grafos genéricos: Implementación

```
/**
 * @post Returns a string representation
 * of this graph.
 */
public String toString() {
    String s = "";
    for (int v = 0; v < V; v++) {
        s += nameOf(v) + ": ";
        for (int w : adj[v]) {
            s += nameOf(w) + " ";
        }
        s += '\n';
    }
    return s;
}
```

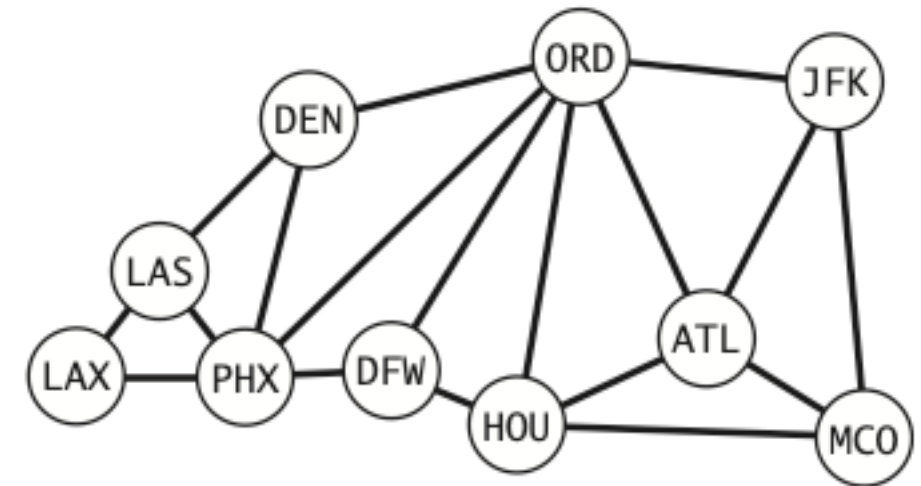


Output toString() :

```
ATL: JFK HOU ORD MCO
DEN: ORD PHX LAS
DFW: PHX ORD HOU
HOU: ORD ATL DFW MCO
JFK: MCO ATL ORD
LAS: DEN LAX PHX
LAX: PHX LAS
MCO: JFK ATL HOU
ORD: DEN HOU DFW PHX JFK ATL
PHX: DFW ORD DEN LAX LAS
```

Grafos genéricos: Implementación

```
/**
 * @post Returns a string representation
 * of this graph.
 */
public String toString() {
    String s = "";
    for (int v = 0; v < V; v++) {
        s += nameOf(v) + ": ";
        for (int w : adj[v]) {
            s += nameOf(w) + " ";
        }
        s += '\n';
    }
    return s;
}
```



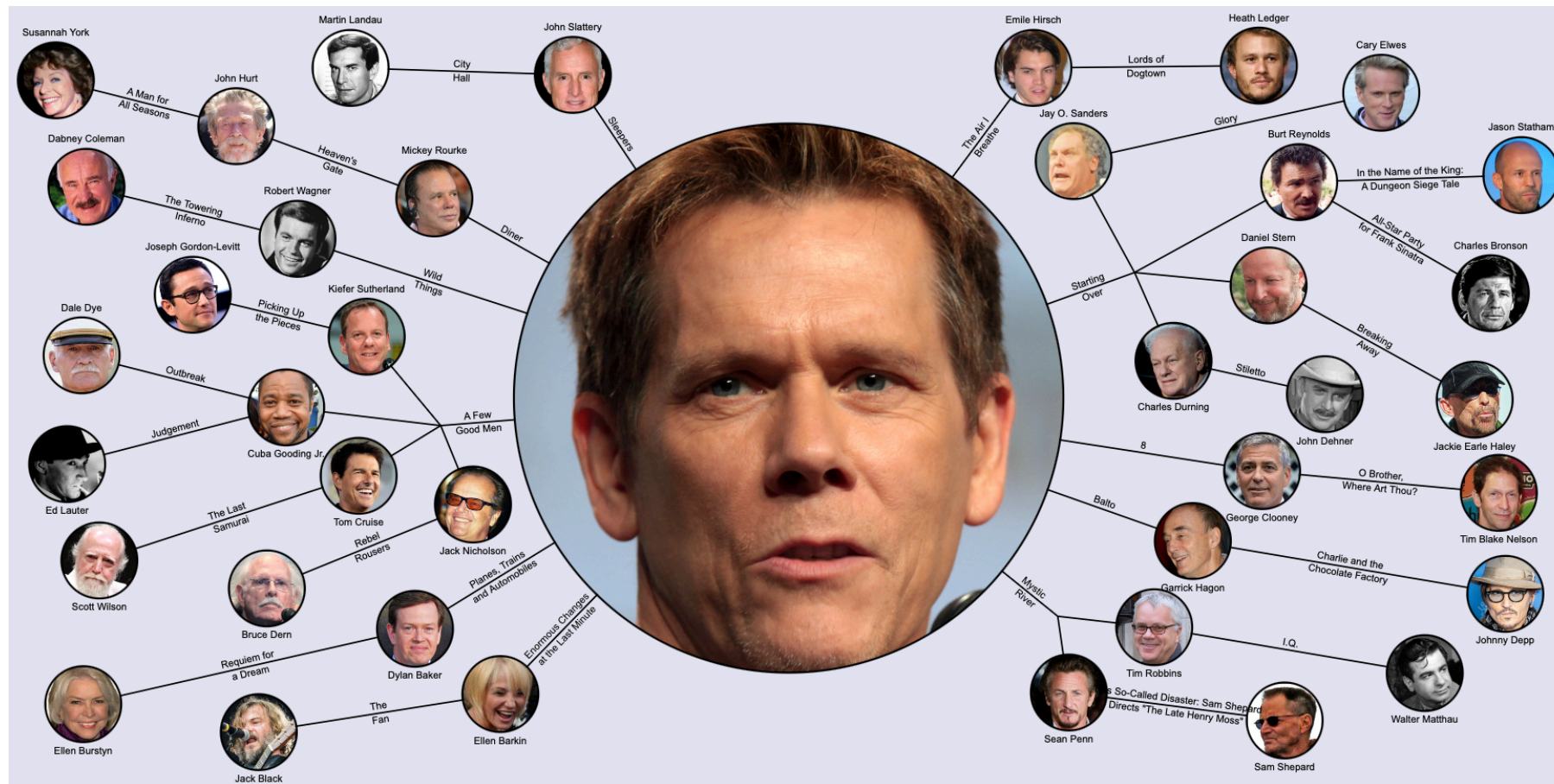
Output toString():

```
ATL: JFK HOU ORD MCO
DEN: ORD PHX LAS
DFW: PHX ORD HOU
HOU: ORD ATL DFW MCO
JFK: MCO ATL ORD
```

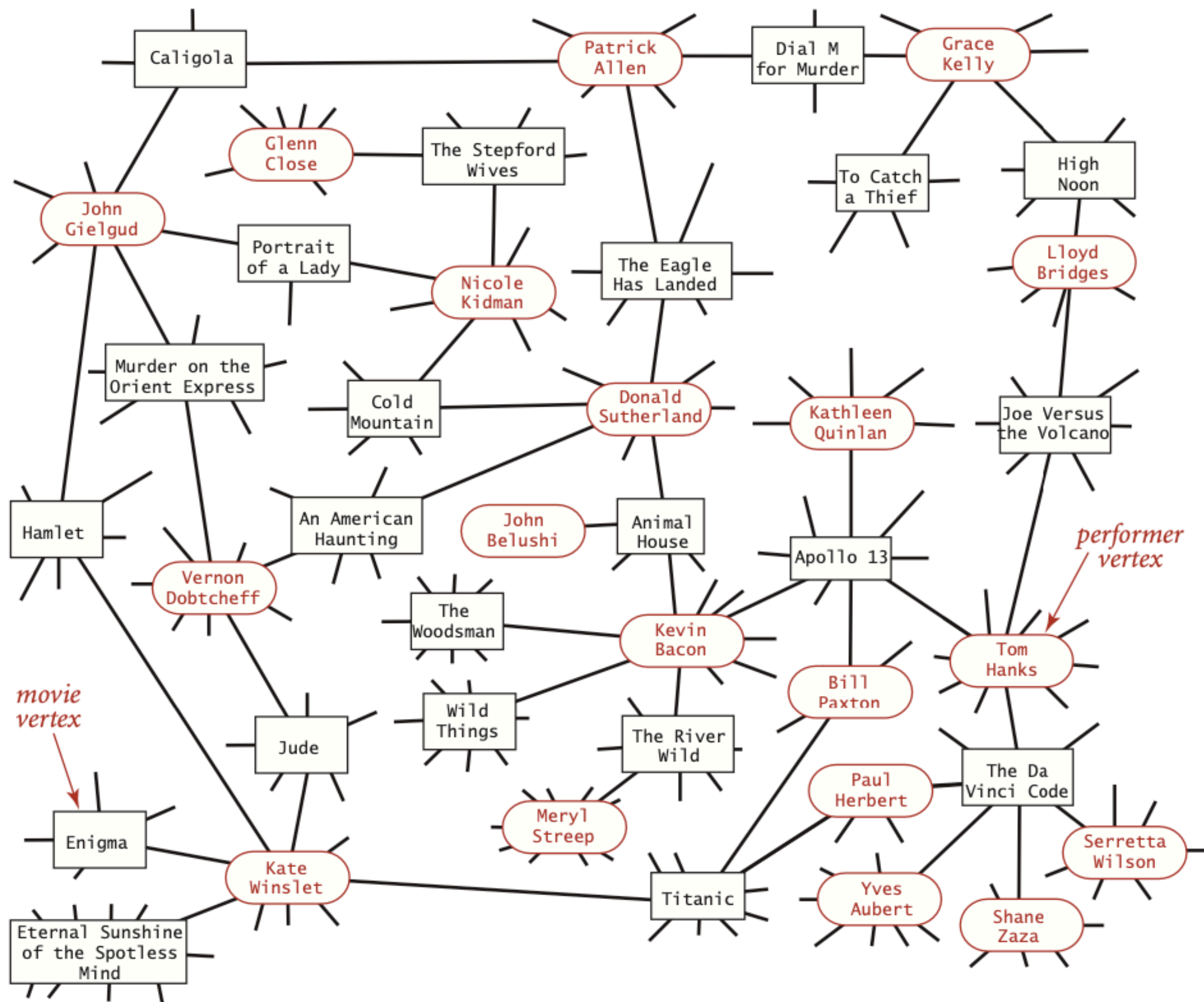
Ejercicio: Implementar grafos genéricos dirigidos, y las versiones dirigidas y no dirigidas de grafos genéricos con matrices de adyacencias

```
JFK ATL
PHX: DFW ORD DEN LAX LAS
```

Ejemplo: Grados de separación de Bacon



- Kevin Bacon es un actor muy prolífico, por lo que el grado de separación entre Bacon y la mayoría de los actores de Hollywood es un número pequeño
- El grado de separación entre Bacon y otro actor se calcula como:
 - Bacon tiene número 0
 - Cualquier actor que comparte elenco en una película con Bacon tiene número 1
 - Tienen número $i+1$ los actores que comparten elenco con cualquier actor que tiene número i



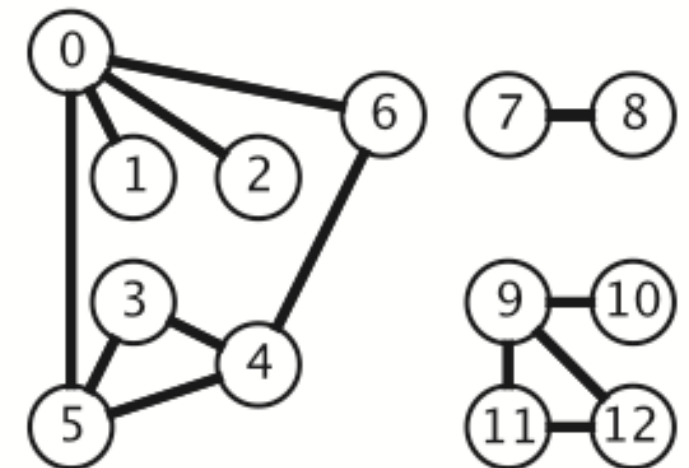
Representación de grafos en archivos

- Muchas veces es conveniente almacenar los grafos en archivos, y que las aplicaciones carguen estos archivos
- Usaremos el siguiente formato para representar grafos en archivos
- La primera línea indica la cantidad de vértices V (opcional)
- La segunda línea indica la cantidad de aristas E (opcional)
- Cada una de las líneas siguientes representa una arista
 - Tienen dos enteros i y j , que representan una arista entre los nodos i y j del grafo

tinyG.txt

$V \rightarrow$ 13
13 $\leftarrow E$

0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3



Ejemplo: Grados de separación de Bacon

- Ejercicio: desarrollar algoritmos para computar los grados de separación entre dos actores cualquiera
 - Y los caminos más cortos (con menor cantidad de aristas) que los conectan
- Para esto, primero se deberá procesar un archivo como el que se muestra a continuación, que tiene una descripción en texto plano del grafo de actores y películas a utilizar, y cargarlo un grafo
 - El archivo será provisto por el equipo docente

movies.txt ← *V and E not explicitly specified*

```
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara...
Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)...
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine
To Be or Not to Be (1942)/Verebes, Ernö (I)/.../Lombard, Carole (I)...
To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...
To Catch a Thief (1955)/París, Manuel/.../Grant, Cary/.../Kelly, Grace/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../ Tucci, Maria...
...
```

movie ↑ *performers* ↑ *"/" delimiter*

Actividades

- Leer el capítulo 22 del libro "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- Implementaciones basadas en el código del repositorio del libro: <https://github.com/kevin-wayne/algs4>
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983