

Estructuras de Datos

Clase 15 – Grafos (Primera Parte)



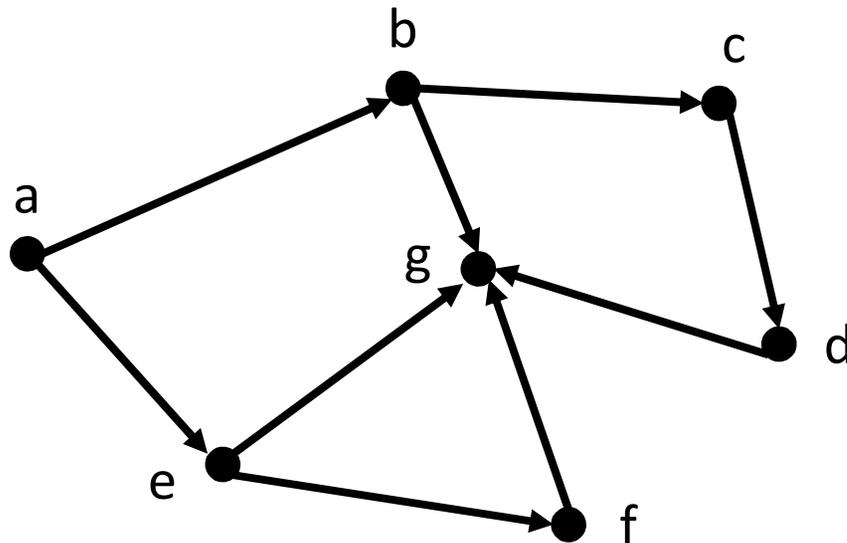
Dr. Sergio A. Gómez
<http://cs.uns.edu.ar/~sag>



Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Definición: Grafo dirigido

- Un grafo $G=(V,E)$ es un conjunto V de vértices (o nodos) y un conjunto $E \subseteq V \times V$ de aristas dirigidas (o arcos dirigidos).
- Intuitivamente E permite representar una relación entre elementos de V .

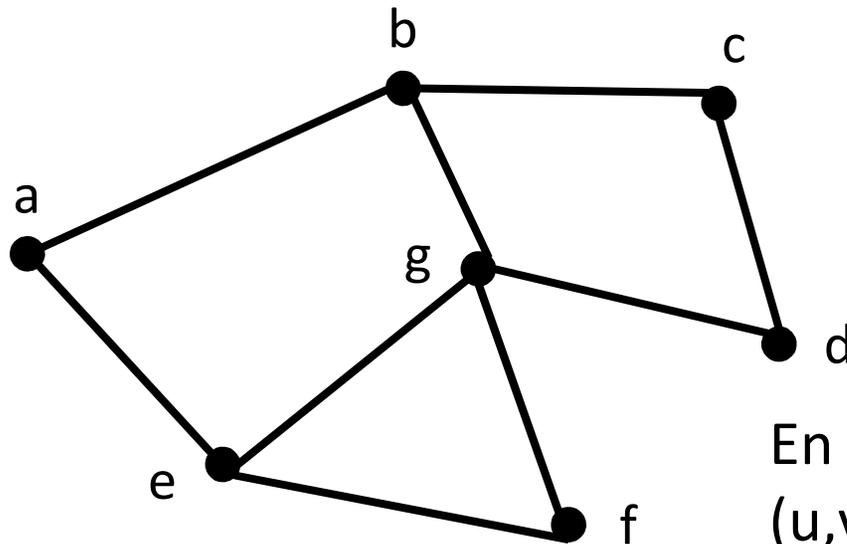


$V = \{ a, b, c, d, e, f, g \}$
 $E = \{ (a,b), (b,c), (c,d), (d,g), (e,g), (e,f), (f,g), (a,e), (b,g) \}$

En el grafo dirigido, el arco (u,v) es un par ordenado

Definición: Grafo no dirigido

- Un grafo $G=(V,E)$ es un conjunto V de vértices (o nodos) y un conjunto $E \subseteq V \times V$ de aristas (o arcos).
- Intuitivamente E permite representar una relación (simétrica) entre elementos de V .



$V = \{ a, b, c, d, e, f, g \}$

$E = \{ (a,b), (b,c), (c,d), (d,g), (e,g), (e,f), (f,g), (a,e), (b,g) \}$

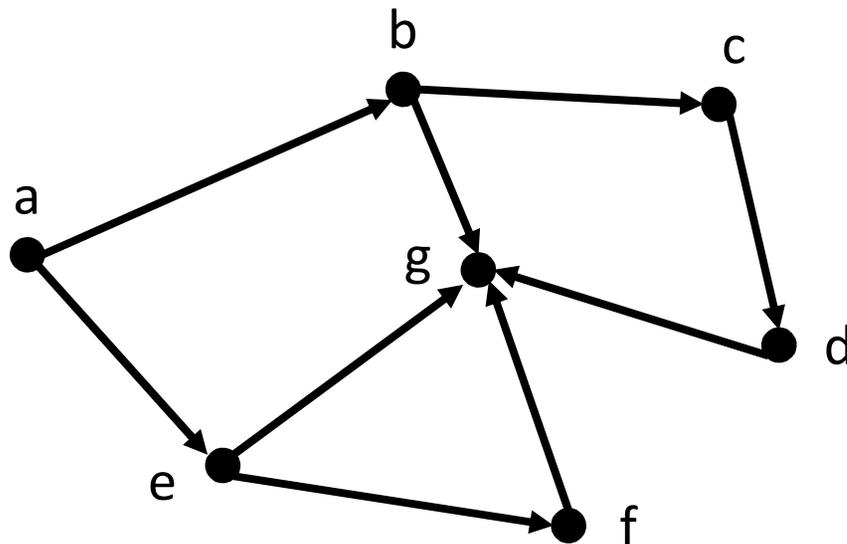
En el grafo no dirigido, el arco (u,v) en realidad es $\{u,v\}$ ya que (u,v) es lo mismo que (v,u) .

- Ejemplo de grafo dirigido: Grafo de herencia de interfaces en Java.
- Ejemplo de grafo dirigido: Grafo donde los nodos son esquinas de una ciudad y los arcos son cuerdas de una mano.
- Ejemplo de grafo dirigido: Grafo donde los nodos son números enteros y los arcos indican qué número divide a otro. E.g. como “2 divide a 6”, habrá un arco (2,6).
- Ejemplo de grafo no dirigido: Grafo de colaboración entre coautores científicos.
- Ejemplo de grafo no dirigido: Grafo donde los nodos son esquinas de un pueblo chico y los arcos son calles doble mano.

Definiciones

- En un grafo no dirigido, se habla de grado porque los arcos que inciden y los que emergen son los mismos.

- Arcos que inciden en un vértice $v = \{ (x,y) \in E : y=v \} = \{ (x,v) : (x,v) \in E \}$
- Arcos que emergen de un vértice $v = \{ (x,y) \in E : x=v \} = \{ (v,y) : (v,y) \in E \}$
- Grado de incidencia/entrada de un vértice v : $\text{indegree}(v) =$ cantidad de arcos que inciden en v
- Grado de emergencia/salida de un vértice v : $\text{outdegree}(v) =$ cantidad de arcos que emergen de v



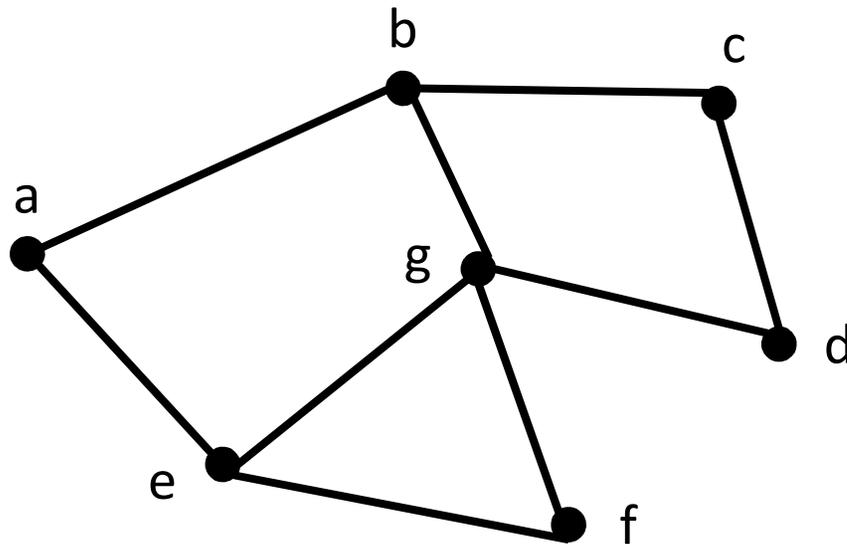
Arcos emergentes de $b = \{ (b,c), (b,g) \}$

$\text{Outdegree}(b) = 2$

Arcos incidente en $g = \{ (b,g), (e,g), (d,g), (f,g) \}$

$\text{Indegree}(g) = 4$

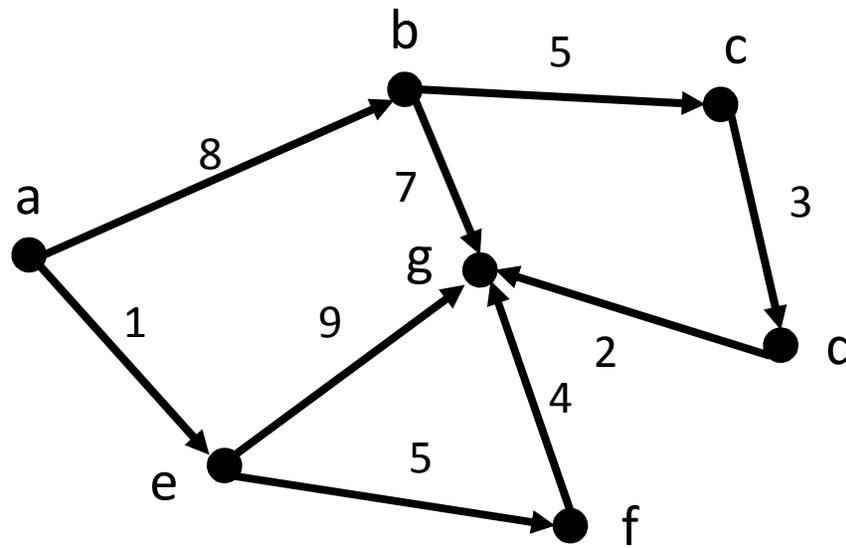
En un grafo no dirigido, se habla de grado porque los arcos que inciden y los que emergen son los mismos.
 $\text{Grado}(v) = \text{cantidad de arcos que inciden en } v$



$\text{Grado}(b) = 3$
 $\text{Grado}(g) = 4$

Definiciones

- Grafo pesado: Grafo donde las aristas contienen una etiqueta (o peso cuando las etiquetas son numéricas).

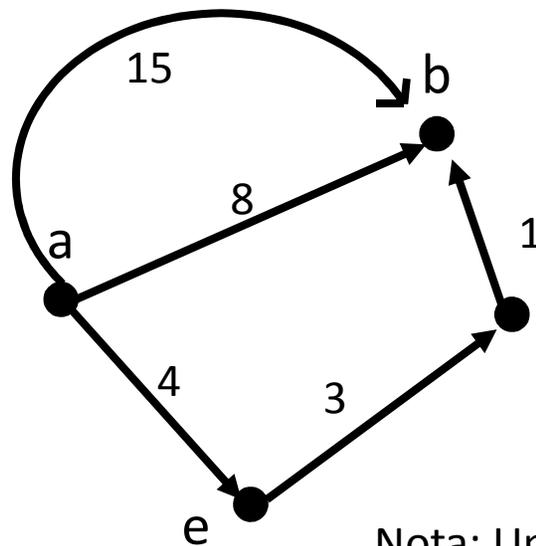


El peso de (c,d) es 3.

- Ejemplo: Digrafo donde los vértices son aeropuertos y los arcos representan vuelos (distancias entre aeropuertos si los pesos son números).

Definiciones

- Multi(di)grafo: (Di)grafo donde el conjunto de arcos es una colección. Es decir puede haber más de un arco entre cada par de vértices (se llaman “arcos paralelos”)
- Ejemplo: Digrafo donde los vértices representan aeropuertos, los arcos son vuelos y puede haber más de un vuelo entre dos aeropuertos.

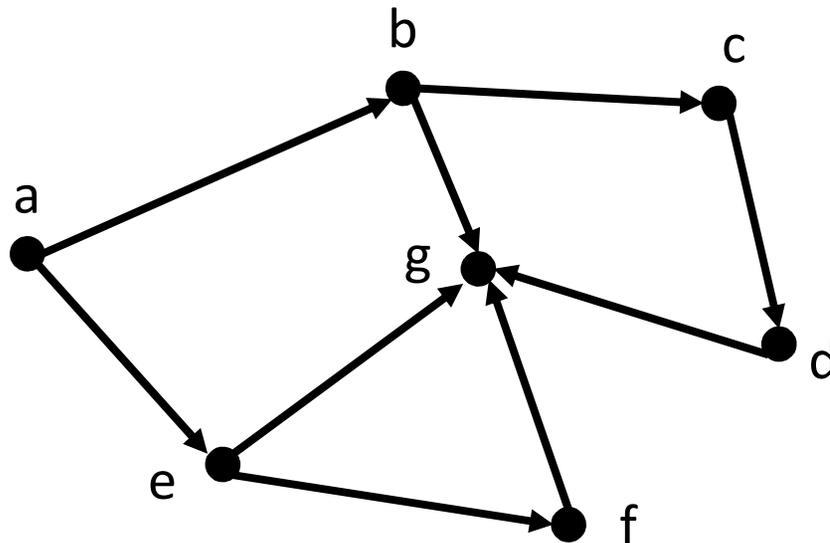


Hay dos arcos que unen “a” con “b”, uno con peso 8 y otro con peso 15.

Nota: Un “grafo simple” es un grafo que no es multigrafo.

Definiciones

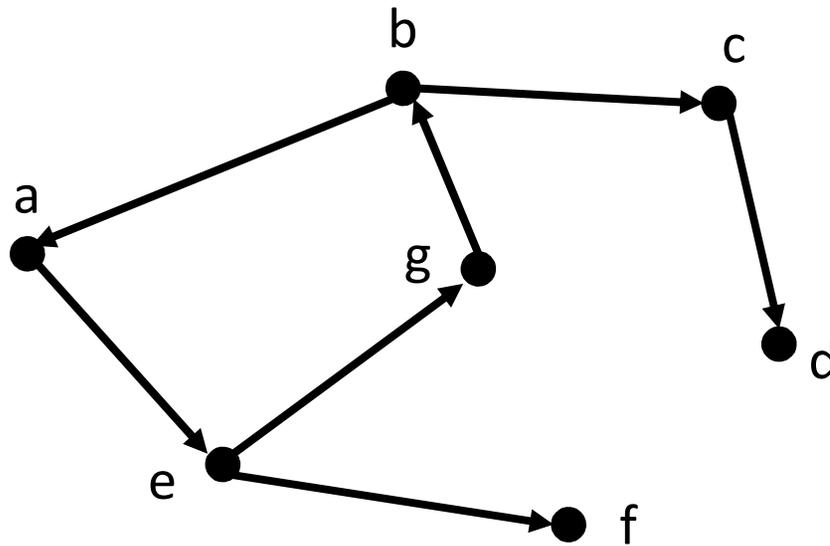
- Camino: Un camino es una secuencia alternante de vértices y arcos tales que empieza en un vértice y termina en otro y cada arco es incidente en sus vértices predecesor y sucesor.



a, b, c, d, g es un camino.
a, e, g es otro camino.

Definiciones

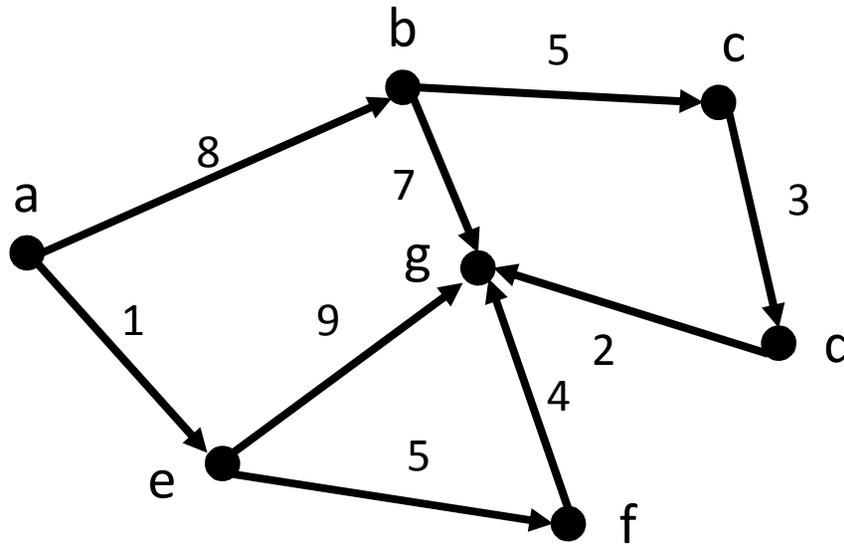
- Ciclo (simple): Un camino que comienza y termina en el mismo vértice (sin repetir arcos).
- Ciclo (camino) dirigido: Un ciclo (camino) donde las aristas tienen dirección y son recorridas en su dirección.



a, e, g, b, a es un ciclo

Definiciones

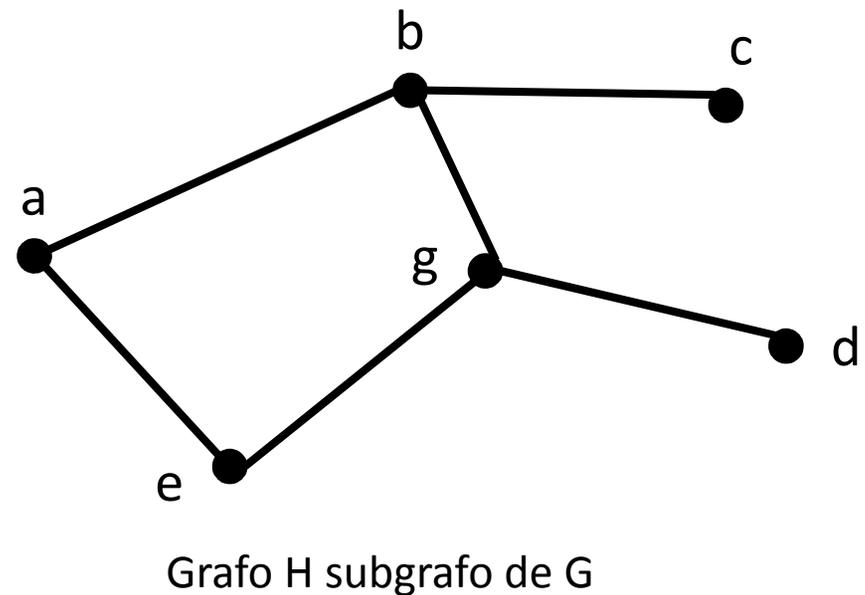
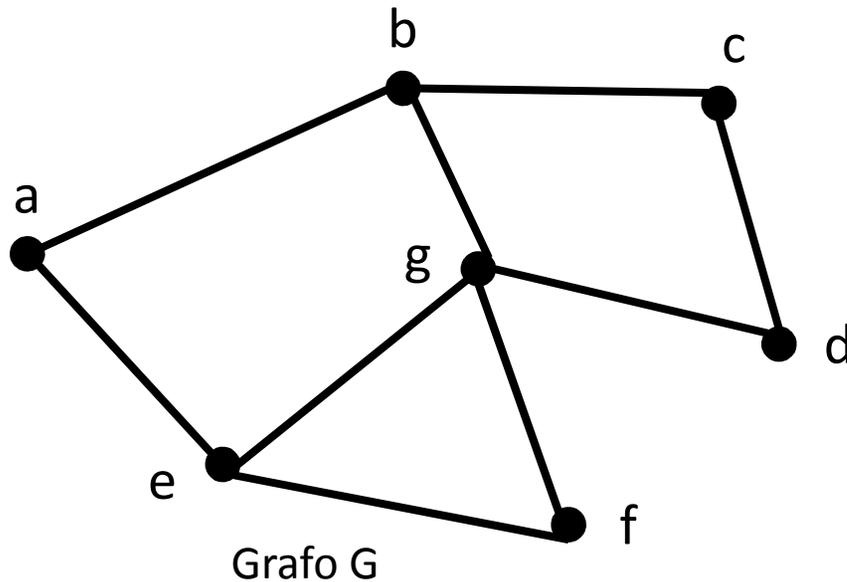
- Costo de un camino (ciclo): Suma de los pesos de los arcos que forman el camino (ciclo)



- Ejemplo: El costo del camino “a, b, c, d, g” es $8+5+3+2=18$.

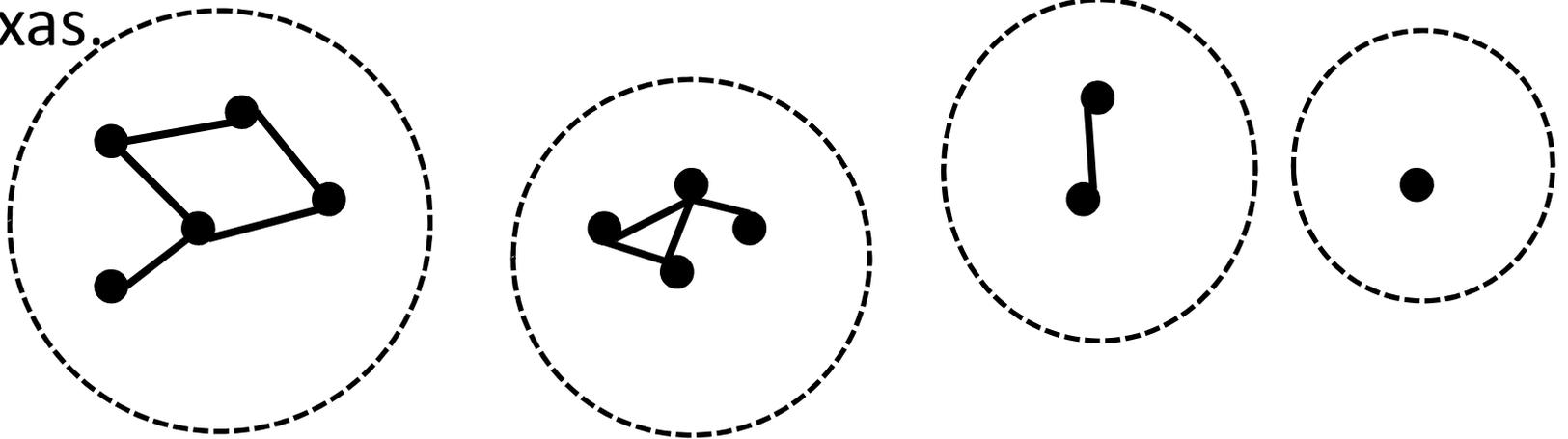
Definiciones

- Subgrafo: Un subgrafo H de un grafo G es un grafo donde los vértices de H son un subconjunto de los vértices de G y los arcos de H son un subconjunto de los arcos de G .



Definiciones

- Grafo conexo: Un grafo G es conexo si para dos vértices cualquiera de G hay un camino entre ellos.
- Componentes conexas: Si un grafo no es conexo, sus subgrafos maximales conexos se llaman componentes conexas.



Este grafo no es conexo porque tiene 4 componentes conexas.

Nota: En grafos dirigidos se habla de “grafo fuertemente conexo” y componente fuertemente conexas porque se pide que los caminos entre vértices sean dirigidos.

ADT Grafo

El tipo de dato abstracto Grafo exporta tres sorts:

- $\text{Graph}\langle V, E \rangle$: Un grafo pesado de vértices con rótulos de tipo V y arcos con rótulos de tipo E
- $\text{Vertex}\langle V \rangle$: La posición de un vértice con rótulo de tipo V
- $\text{Edge}\langle E \rangle$: La posición de un arco con rótulo de tipo E

ADT Grafo

- `vertices()`: Retorna una colección iterable con todos los vértices del grafo.
- `edges()`: Retorna una colección iterable con todos los arcos del grafo.
- `incidentEdges(v)`: Retorna una colección iterable con todos los arcos incidentes sobre un vértice v
- `emergentEdges(v)`: Retorna una colección iterable con todos los arcos emergentes de un vértice v (no está en GT, también le podemos decir `successorEdges(v)`)

ADT Grafo

- `opposite(v,e)`: Retorna el otro vértice w del arco $e=(v,w)$; ocurre un error si e no es incidente (o emergente de v).
- `endVertices(e)`: Retorna un arreglo (de 2 componentes) conteniendo los vértices del arco e .
- `areAdjacent(v,w)`: Testea si los vértices v y w son adyacentes.

ADT Grafo

- `replace(v,x)` : Reemplaza el rótulo del vértice v con x
- `replace(e,x)`: Reemplaza el rótulo del arco e con x
- `insertVertex(x)`: Inserta y retorna un nuevo vértice con rótulo x
- `insertEdge(v, w, x)`: Inserta un arco con rótulo x entre los vértices v y w
- `insertDirectedEdge(v, w, x)`: Inserta un arco dirigido con rótulo x entre los vértices v y w

ADT Grafo

- `removeVertex(v)`: Elimina el vértice v y todos sus arcos adyacentes y retorna el rótulo de v
- `removeEdge(e)`: Elimina el arco e y retorna el rótulo almacenado en e .

ADT Grafo (dirigido)

```
Graph<String, Integer> g = new Digrafo<String, Integer>();
```

```
Vertex<String> bb = g.insertVertex("Bahia Blanca");
```

```
Vertex<String> pa = g.insertVertex("Punta Alta");
```

```
Vertex<String> ba = g.insertVertex("Buenos Aires");
```

```
Vertex<String> mdp = g.insertVertex("Mar del Plata");
```

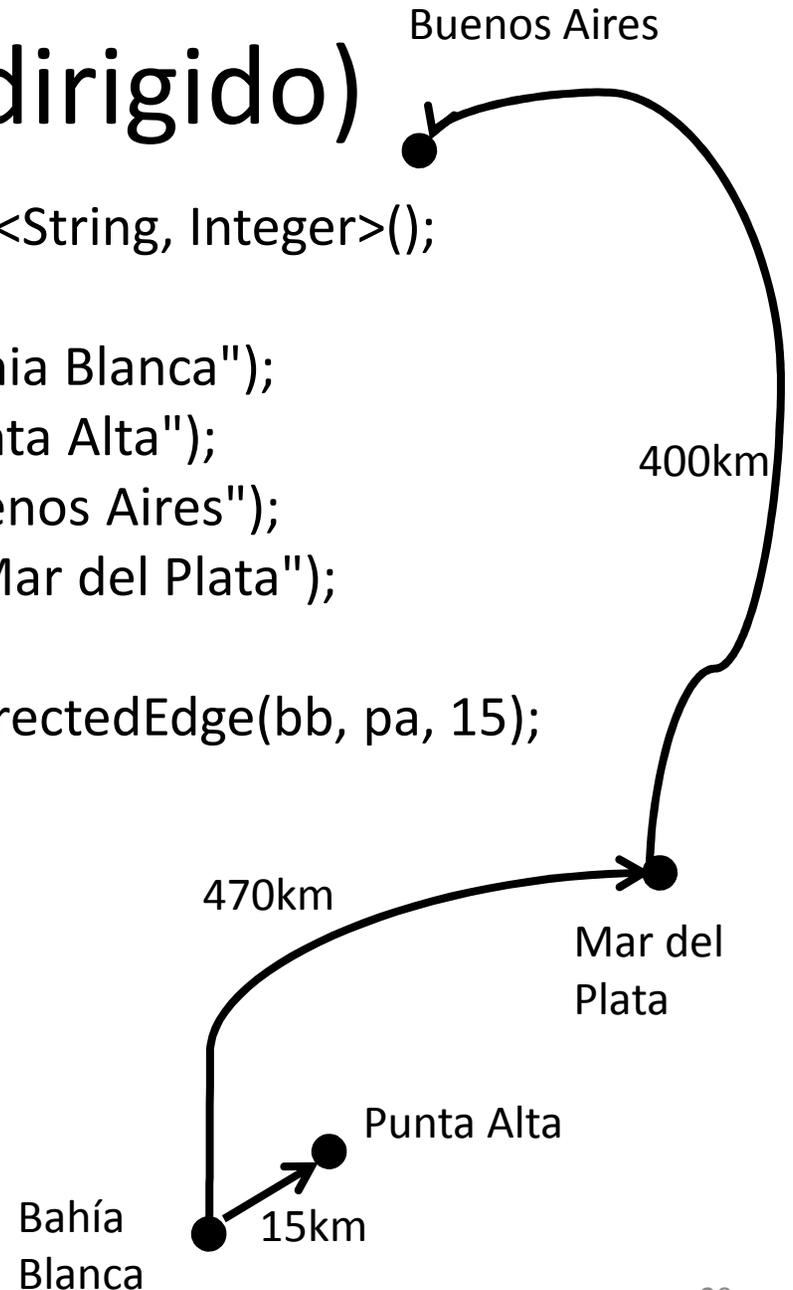
```
Edge<Integer> vueloBB2PA = g.insertDirectedEdge(bb, pa, 15);
```

```
g.insertDirectedEdge(bb, mdp, 470);
```

```
g.insertDirectedEdge(mdp, ba, 400);
```

```
g.removeEdge(vueloBB2PA);
```

```
g.removeVertex(pa);
```



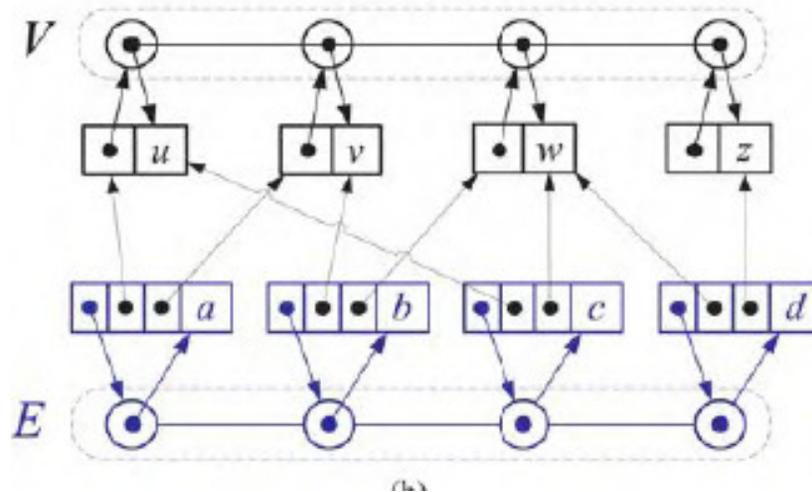
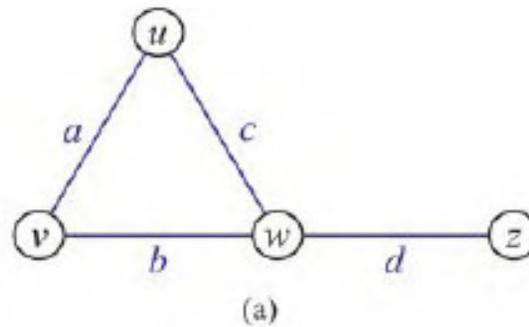
Estructuras de datos para grafos (simples y pesados)

- Lista de arcos: el grafo es una lista de vértices y una lista de arcos (los vértices y los arcos conocen sus rótulos respectivos).
- Lista de adyacencias: El grafo conoce una lista de vértices (opcionalmente también una lista de arcos) y cada vértice conoce su rótulo y los arcos que emergen de él. Los arcos conocen los vértices que unen y su peso.
- Matriz de adyacencias: el grafo es una lista de vértices (opcionalmente conoce también una lista de arcos) y una matriz donde cada componente (i,j) almacena un arco.

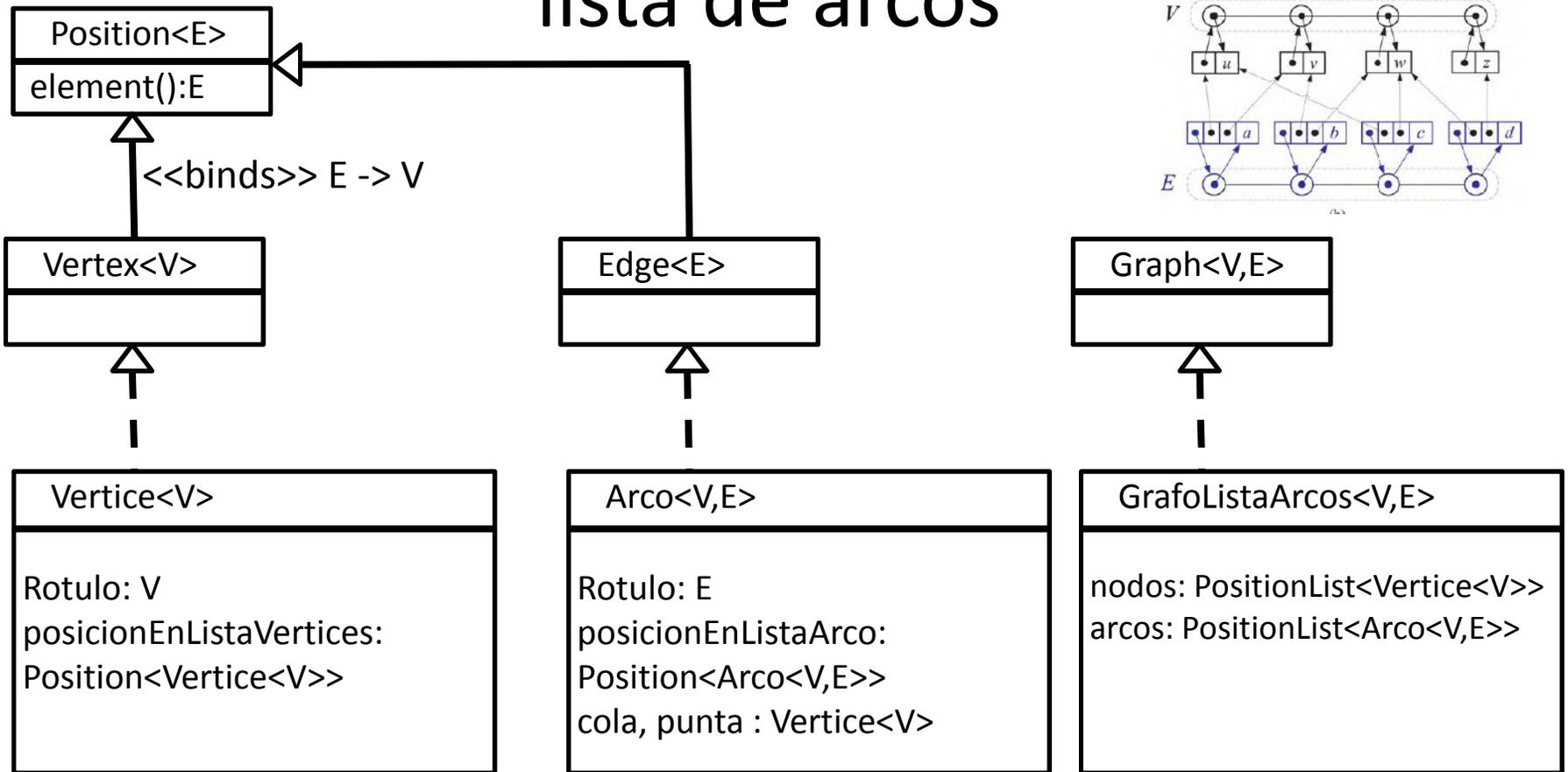
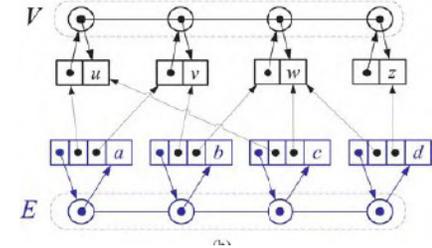
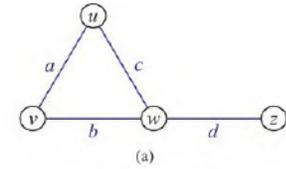
Lista de arcos: El grafo conoce una lista de vértices y una lista de arcos.

Los vértices conocen sus rótulos respectivos y su posición en la lista de vértices.

Los arcos conocen su peso, los vértices que unen y su posición en la lista de arcos.



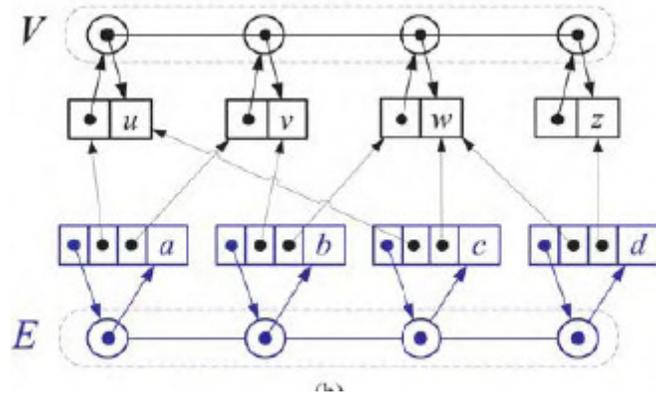
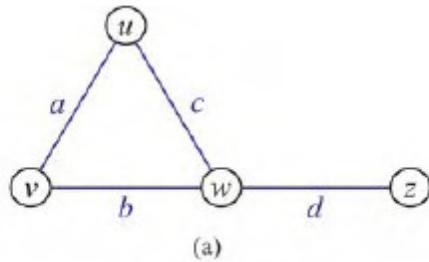
Implementación de lista de arcos



Performance deseada de la lista de arcos

Dado un grafo $G=(V,A)$

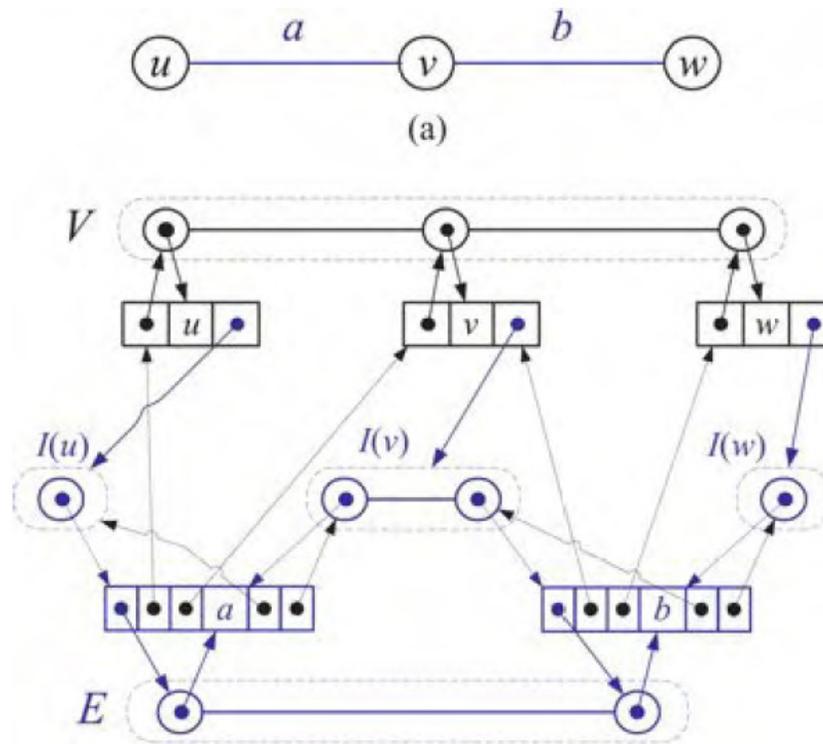
Sea $n = \#V$ y $m = \#A$



Operación	Tiempo
vertices()	$O(n)$
edges()	$O(m)$
endVertices(e), opposite(v,e)	$O(1)$
incidentEdges(v), areAdjacent(v,w)	$O(m)$
replace(v,x), replace(e,x)	$O(1)$
insertVertex(x), insertEdge(v,w,x), removeEdge(e)	$O(1)$
removeVertex(v)	$O(m)$

Lista de adyacencias para un grafo dirigido $G=(V,E)$ de acuerdo a GT

La mejora que propone la lista de adyacencias es introducir un lista que para cada vértice indica los arcos que emergen cuando el grafo es dirigido y los que emergen/inciden cuando el grafo es no dirigido. El libro solo grafica los grafos no dirigidos:



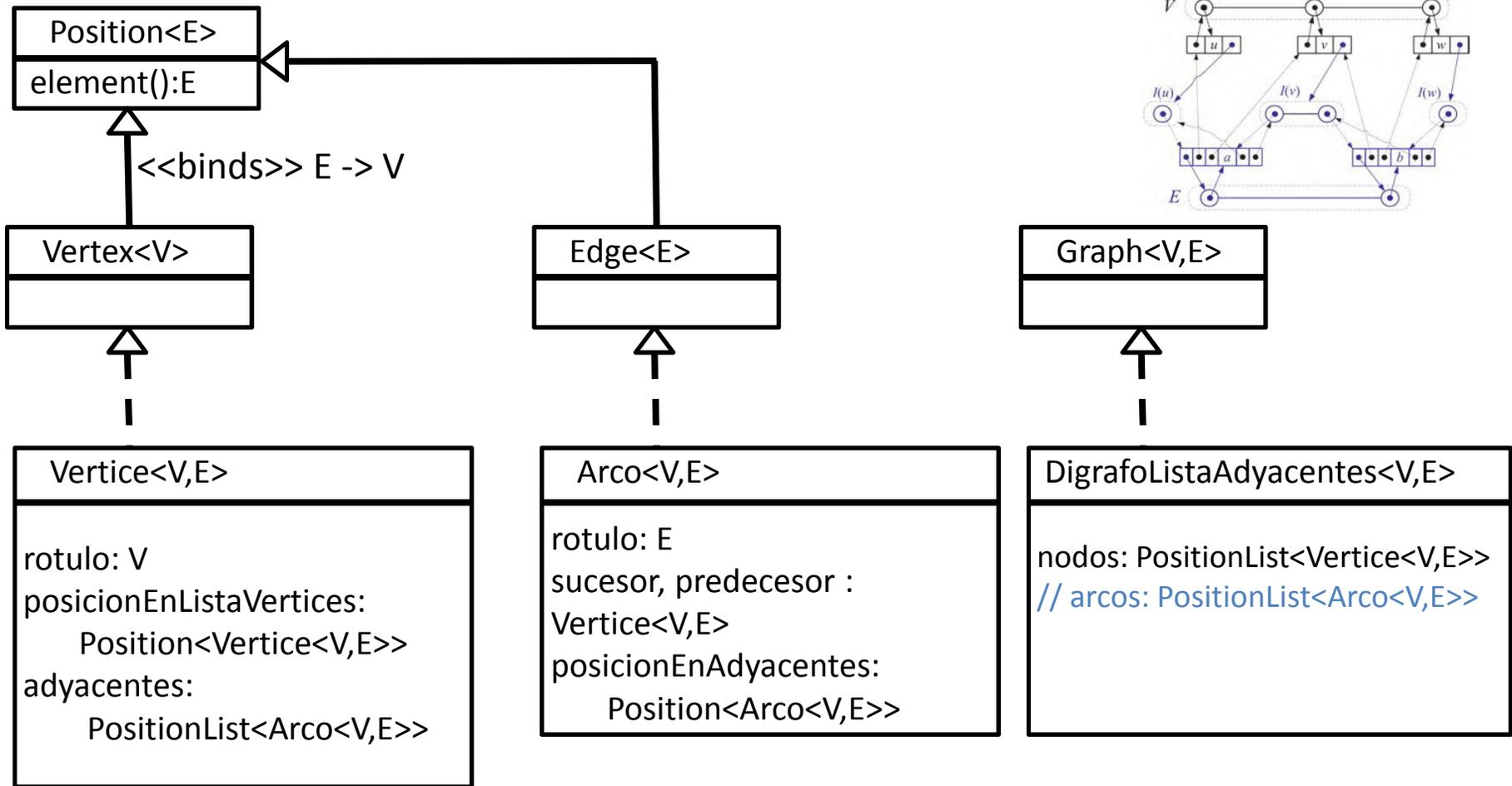
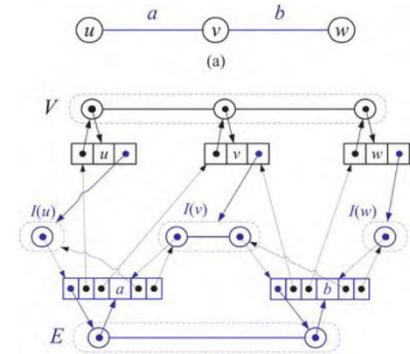
Implementación de digrafo con lista de adyacencias

```
public interface Vertex<E> extends Position<E> { } // Vertex.java (luego lo mejoraremos)
```

```
public interface Edge<E> extends Position<E> { } // Edge.java (luego lo mejoraremos)
```

```
public interface Graph<V, E> { // Graph.java
    public Iterable<Vertex<V>> vertices();
    public Iterable<Edge<E>> edges();
    // public Iterable<Edge<E>> incidentEdges(Vertex<V> v); // Para no dirigidos [GT]
    public Iterable<Edge<E>> emergentEdges(Vertex<V> v); // No está en GT
    public Vertex<V> opposite(Vertex<V> v, Edge<E> e);
    public Vertex<V> [] endVertices(Edge<E> e);
    public boolean areAdjacent(Vertex<V> v, Vertex<V> w);
    public V replace( Vertex<V> v, V x);
    public E replace( Edge<E> e, E x);
    public Vertex<V> insertVertex(V x);
    public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E x);
    public V removeVertex(Vertex<V> v);
    public E removeEdge(Edge<E> e);
}
```

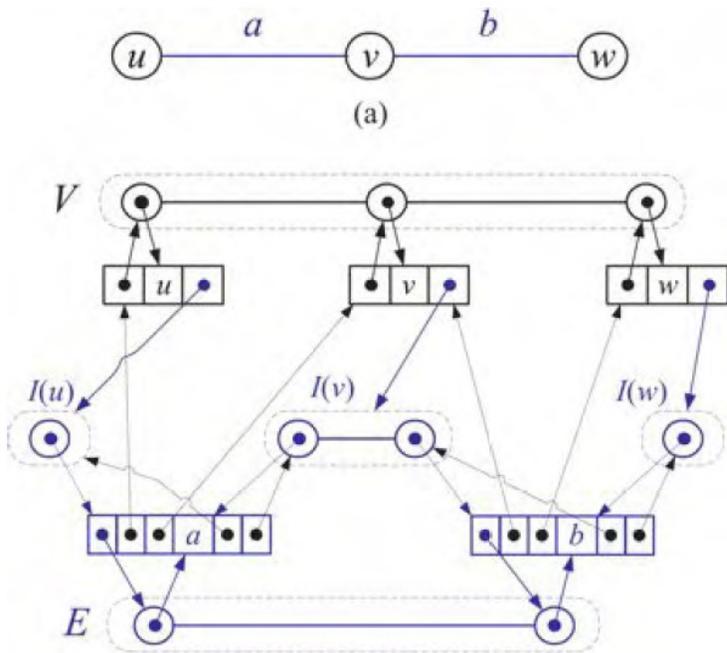
Implementación de grafo dirigido con lista de adyacencias



NOTA: La lista de arcos no es estrictamente necesaria. El libro sí la usa. Agregarla queda como ejercicio, lo cual requiere modificar el código presentado a continuación.

NOTA: El libro usa la misma notación para el grafo $G=(V,E)$ y para los tipos de los rótulos de vértice V y rótulos de arco E .

Performance deseada de la lista de adyacencias de acuerdo a GT



Operación	Tiempo
vertices()	$O(n)$
edges()	$O(m)$
endVertices(e), opposite(v,e)	$O(1)$
emergentEdges(v), areAdjacent(v,w)	$O(\text{deg}(v))$
replace(v,x), replave(e,x)	$O(1)$
insertVertex(x), insertEdge(v,w,x), removeEdge(e)	$O(1)$
removeVertex(v)	$O(\text{deg}(v))$

// Archivo: Vertice.java (o también se puede hacer clase anidada privada de la clase Digrafo.java que implementa Graph)

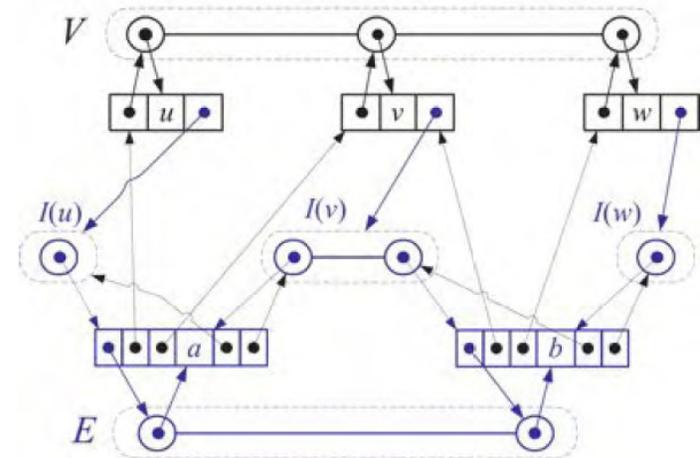
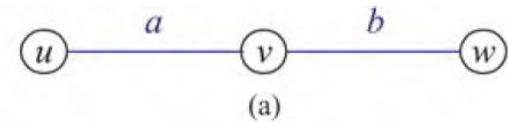
```
public class Vertice<V,E> implements Vertex<V> {
private V rotulo;
private PositionList<Arco<V,E>> adyacentes;
private Position<Vertice<V,E>> posicionEnNodos;
```

```
public V element() { return rotulo; }
```

```
public Vertice( V rotulo ) {
    this.rotulo = rotulo;
    adyacentes = new Lista<Arco<V,E>>();
}
```

// Setters y getters

```
public void setRotulo(V nuevoRotulo) { ... }
public PositionList<Arco<V,E>> getAdyacentes() { ... }
public void
setPosicionEnNodos(Position<Vertice<V,E>> p ) { ... }
public Position<Vertice<V,E>> getPosicionEnNodos()
{ ... }
}
```

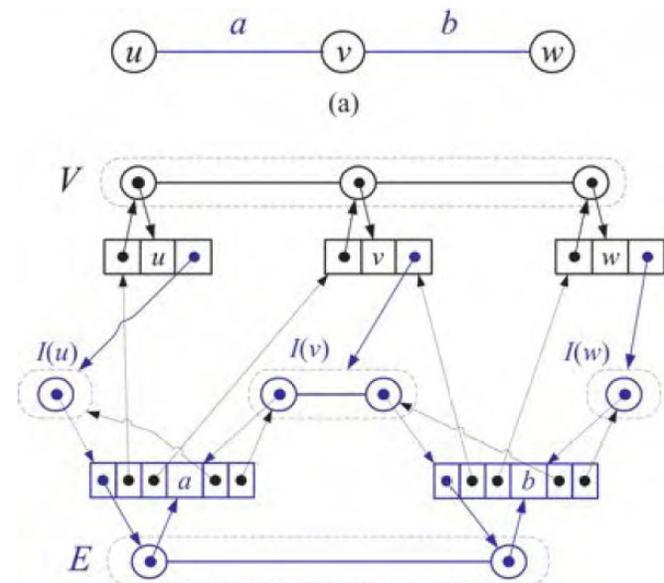


// Archivo Arco.java: También se puede hacer clase anidada y privada de la clase Digrafo que implementa Graph.

```

public class Arco<V,E>
    implements Edge<E> {
private E rotulo;
private Vertice<V,E> sucesor, predecesor;
private Position<Arco<V,E>> posicionEnAdyacentes;
public Arco( E rotulo,
    Vertice<V,E> predecesor,
    Vertice<V,E> sucesor )
{ /* solo setea atributos */ }
public E element() { return rotulo; }
public Vertice<V,E> getPredecesor() { ... }
public Vertice<V,E> getSucesor() { ... }
public Position<Arco<V,E>>
getPosicionEnAdyacentes() { ... }
public void setPosicionEnAdyacentes(
    Position<Arco<V,E>> p) { ... }
}

```



// Digrafo.java

```
public class Digrafo<V,E> implements Graph<V,E>
{
    protected PositionList<Vertice<V,E>> nodos;

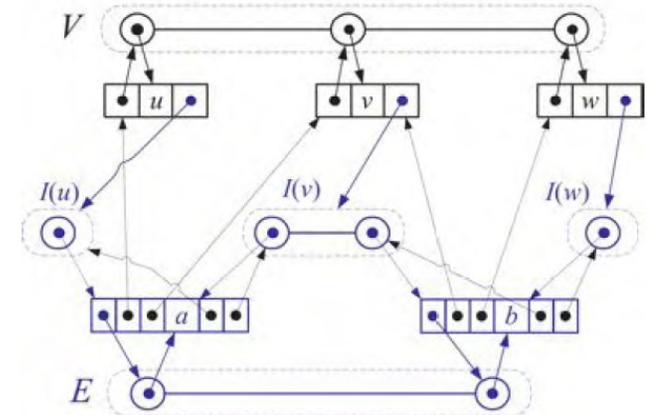
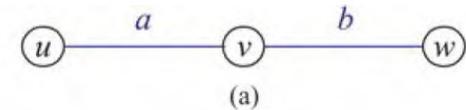
    public Digrafo() {
        nodos = new Lista<Vertice<V,E>>();
    }

    public Iterable<Vertex<V>> vertices() {
        PositionList<Vertex<V>> lista =
            new Lista<Vertex<V>>();

        for( Vertex<V> v : nodos )
            lista.addLast(v);

        return lista;
    }
}

Tvertices(n, m) = O(n)
```



Uso:

```
for( Vertex<V> v : g.vertices() )
    System.out.println( v.element() );
```

Tarea: Pensar cómo agregar la lista de arcos.

```
// Recordar que los atributos del grafo son:
//   protected PositionList<Vertice<V,E>> nodos;
```

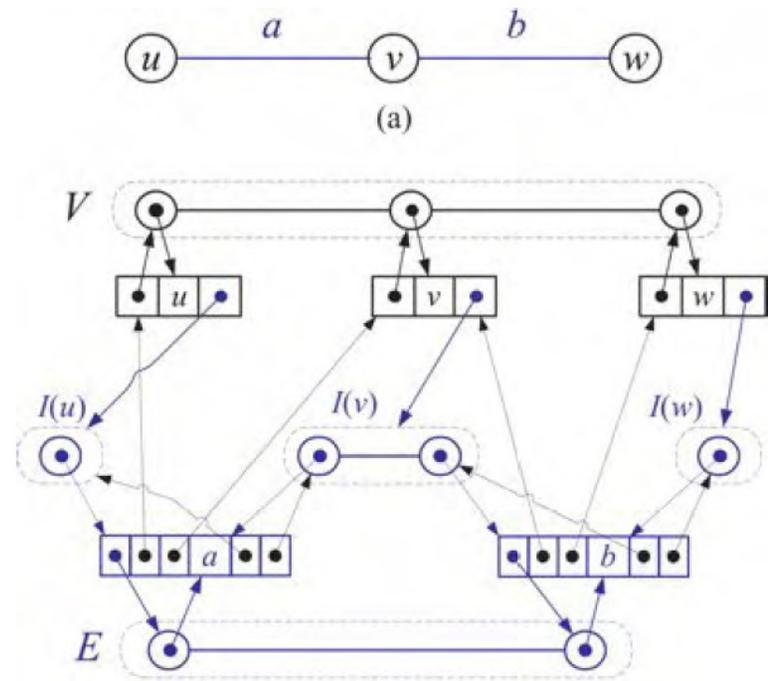
```
public Iterable<Edge<E>> edges() {
    PositionList<Edge<E>> lista =
        new ListaDoblementeEnlazada<Edge<E>>();

    for( Vertex<V> v : nodos )
        for( Edge<E> e : emergentEdges(v))
            lista.addLast(e);

    return lista;
}
```

$T_{\text{edges}}(n, m) = O(n+m)$

Tarea: Si tengo la lista de arcos,
solo tengo que recorrerla (es una optimi-
zación propuesta por el libro).

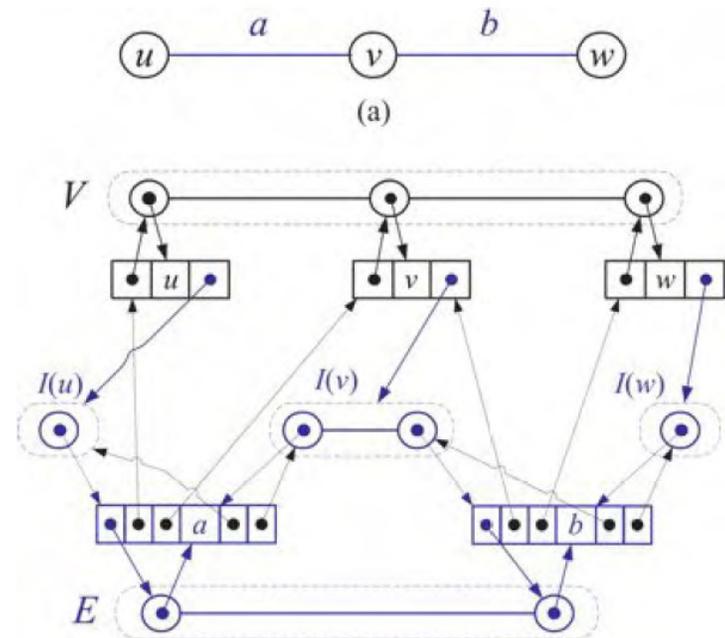


```
// Atributos del grafo:
// protected PositionList<Vertice<V,E>> nodos;
```

```
public Iterable<Edge<E>> emergentEdges( Vertex<V> v) {
PositionList<Edge<E>> lista = new Lista<Edge<E>>();
```

```
Vertice<V,E> vert = (Vertice<V,E>) v;
for( Edge<E> e : vert.getAdyacentes() )
    lista.addLast(e);
return lista;
}
```

$T_{\text{emergentEdges}}(n,m) = O(\text{deg}(v))$

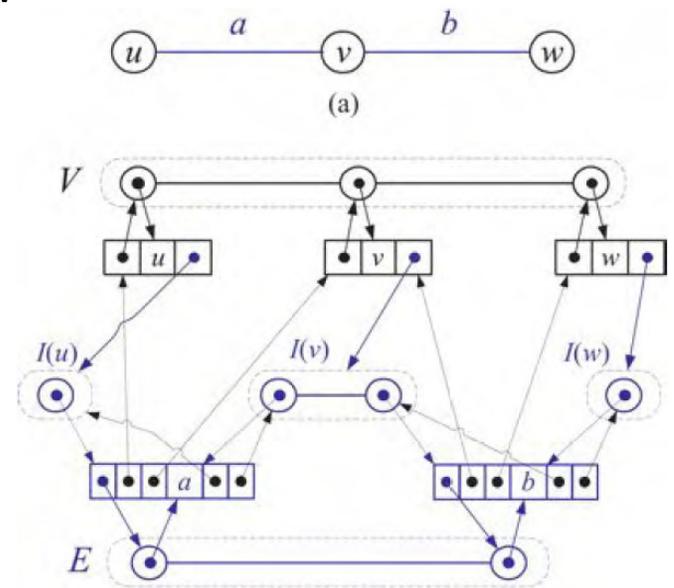


```
// Atributos del grafo:
// protected PositionList<Vertice<V,E>> nodos;
```

```
public Vertice<V> insertVertex(V x) {
    Vertice<V,E> v = new Vertice<V,E>(x);
    nodos.addLast(v);

    v.setPositionEnNodos(nodos.last());
    return v;
}
```

$$T_{\text{insertVertex}}(n,m) = O(1)$$

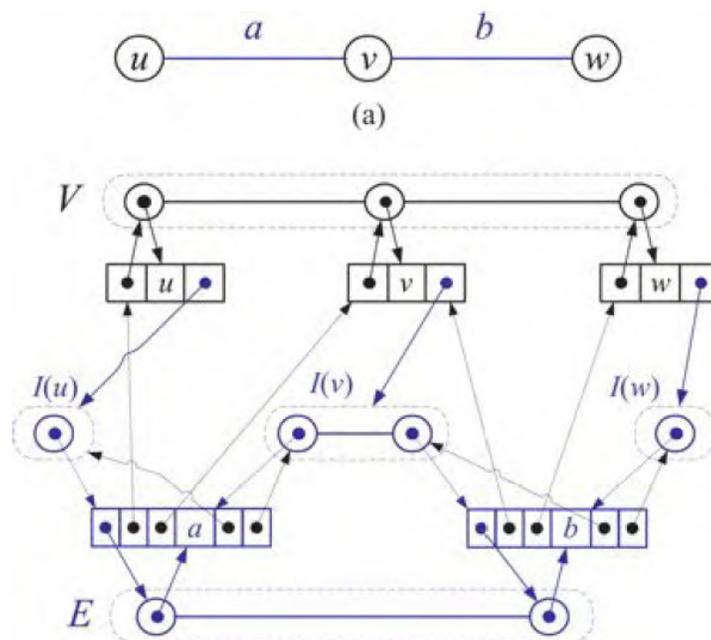


```
// Atributos del grafo:
//   protected PositionList<Vertice<V,E>> nodos;
// Inserta un arco dirigido de v a w
```

```
public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E x) {
    Vertice<V,E> vv = (Vertice<V,E>) v;
    Vertice<V,E> ww = (Vertice<V,E>) w;
    Arco<V,E> arco = new Arco<V,E>( x, vv, ww );
    vv.getAdyacentes().addLast( arco );
    arco.setPosicionEnAdyacentes( vv.getAdyacentes().last() );
    return arco;
}
```

$T_{\text{insertEdge}}(n,m) = O(1)$

Discusión: ¿Qué hay que cambiar aquí para lograr la implementación de grafo no dirigido propuesta por el libro?

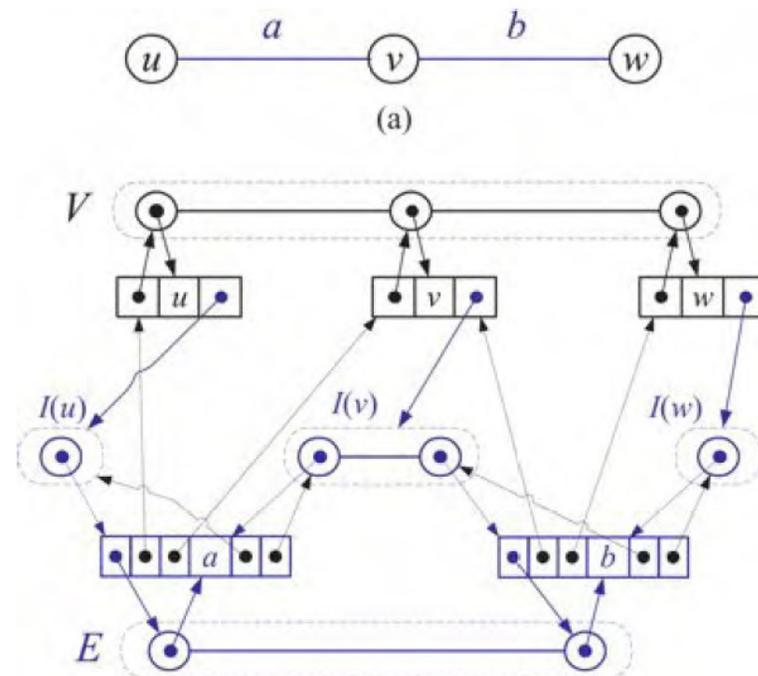


```

// Atributos del grafo:
//   protected PositionList<Vertice<V,E>> nodos;
public E removeEdge(Edge<E> e) {
  try {
    Arco<V,E> ee = (Arco<V,E>) e;
    Position<Arco<V,E>> pee = ee.getPosicionEnAdyacentes();
    Vertice<V,E> pred = ee.getPredecesor();
    return pred.getAdyacentes().remove(pee).element();
  } catch(InvalidPositionException e) {
    e.printStackTrace();
    return null;
  }
}

```

$T_{\text{removeEdge}}(n,m) = O(1)$



```

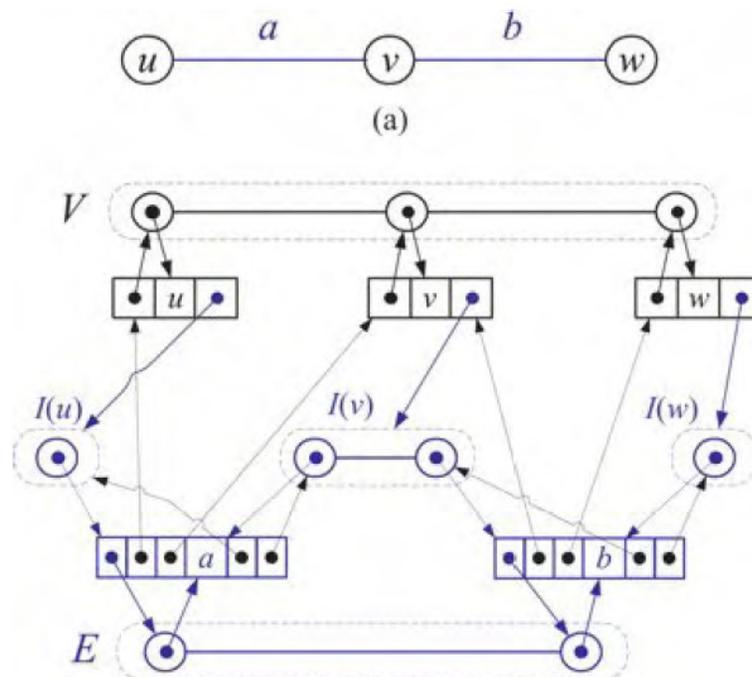
// Atributos del grafo:
//   protected PositionList<Vertice<V,E>> nodos;

// Precondición: Asume que v no tiene arcos adyacentes.
public V removeVertex(Vertex<V> v) {
try {
    Position<Vertice<V,E>> pos = ((Vertice<V,E>) v).getPosicionEnNodos();
    return nodos.remove( pos ).element();
} catch(InvalidPositionException e) {
    e.printStackTrace();
    return null;
}
}

```

$T_{\text{removeVertex}}(n,m) = O(1)$

Discusión: Pensar cómo modificarla para primero eliminar todos los arcos que salen del nodo v (como propone GT)



Bibliografía

- Capítulo 13 de M. Goodrich & R. Tamassia, Data Structures and Algorithms in Java. Fourth Edition, John Wiley & Sons, 2006.