Estructuras de Datos
Clase 16 – Grafos (Segunda 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

ADT Grafo

El tipo de dato abstracto Grafo exporta tres sorts:

- Graph<V,E>: Un grafo pesado de vértices con rótulos de tipo V y arcos con rótulos de tipo E
- Vertex<V>: La posición de un vértice con rótulo de tipo V
- Edge<E>: 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

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

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.

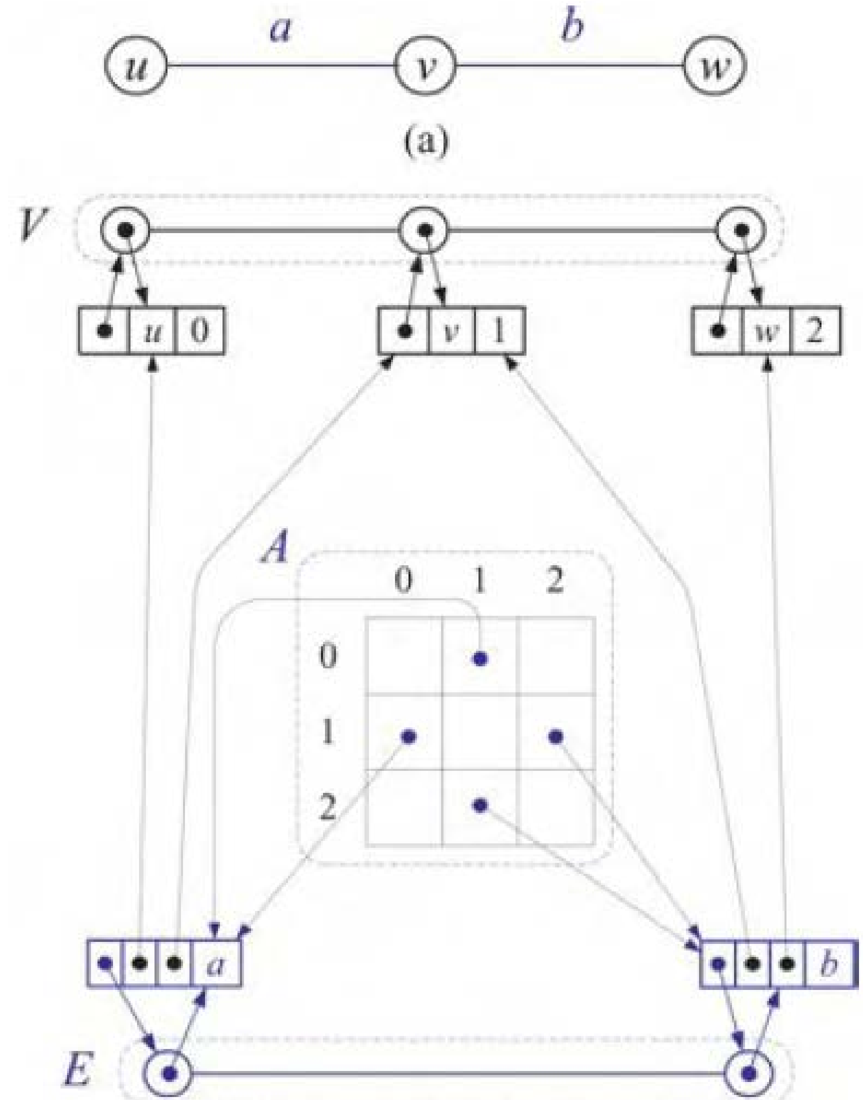
ADT Grafo

```
Graph<String, Integer> g = new GrafoConMatriz<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.insertEdge(bb, pa, 15);
g.insertEdge(bb, mdp, 470);
g.insertEdge(mdp, ba, 400);

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

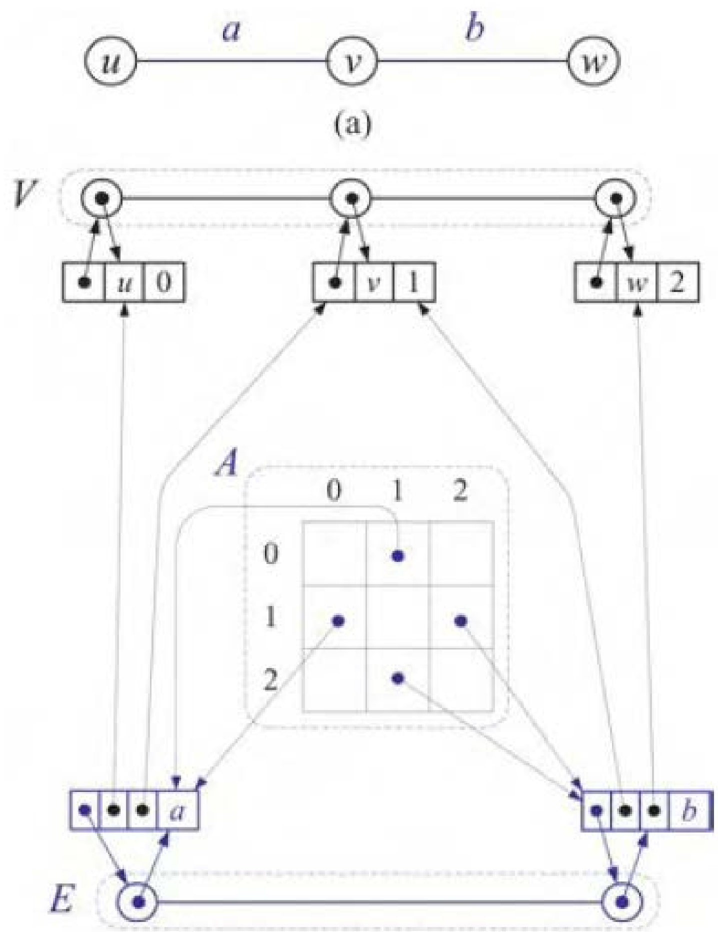
Estructuras de datos para grafos

- Lista de arcos
- Lista de adyacencias
- Matriz de adyacencias

Matriz de adyacencias



Performance de matriz de adyacencias

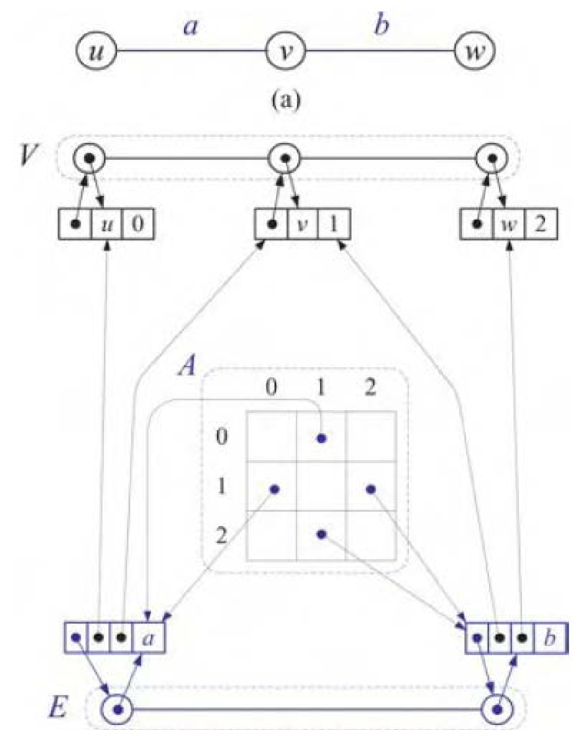


Operación	Tiempo
vertices()	O(n)
edges()	O(m)
endVertices(e), opposite(v,e), areAdjacent(v,w)	O(1)
incidentEdges(v)	O(n+deg(v))
replace(v,x), replace(e,x), insertEdge(v,w,x), removeEdge(e)	O(1)
insertVertex, removeVertex	O(n^2)

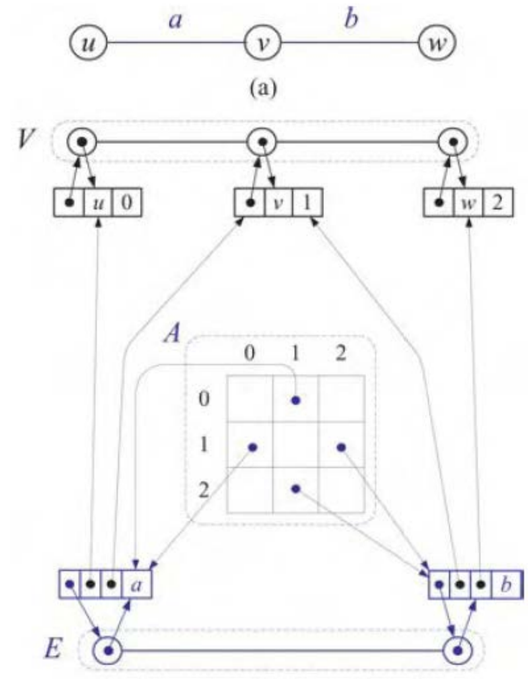
¿Por qué insertVertex y removeVertex tienen O(n^2)?
¿Cuál es el T(n) de removeVertex(v) si asumimos que no hay arcos que salen o llegan a v?
En removeVertex(v), ¿qué podría hacer para tener O(1) si v corresponde a una fila en el medio de la matriz?

Implementación de grafo no dirigido con matriz de adyacencias

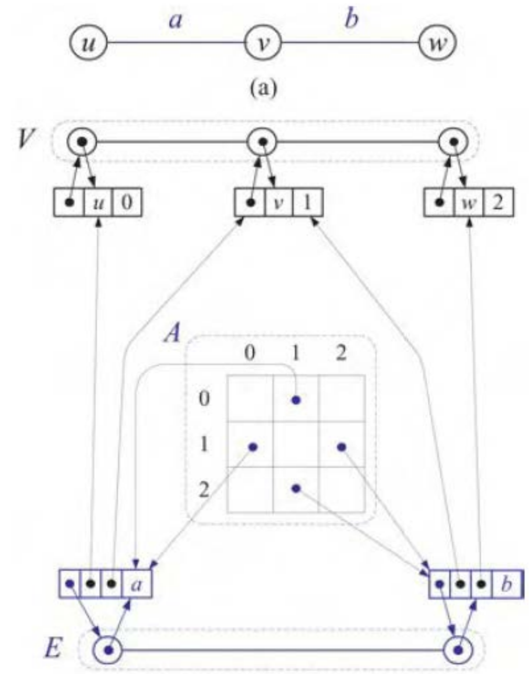
```
public class GrafoConMatriz<V,E> implements Graph<V,E> {

        protected PositionList<Vertex<V>> vertices;
        protected PositionList<Edge<E>> arcos;
        protected Edge<E> [][] matriz;
        protected int cantidadVertices;
```



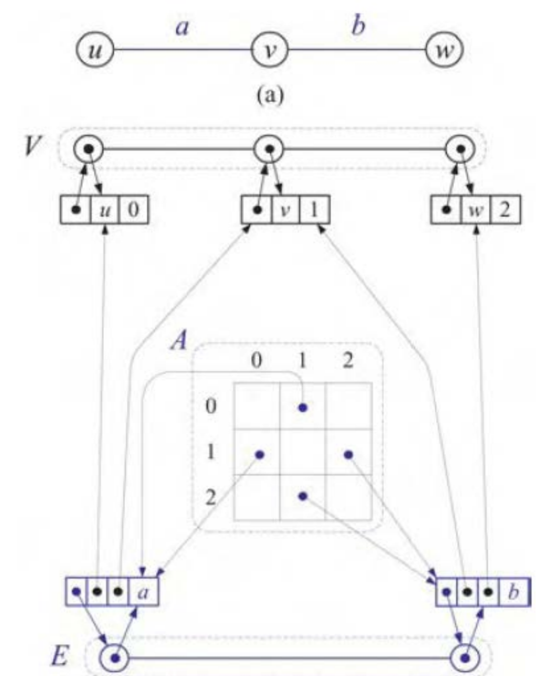
```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

private class Vertice<V> implements Vertex<V> {
  private V rotulo;
  private int indice;
  private Position<Vertex<V>> posicionEnVertices;

  public Vertice( V rotulo, int indice ) { this.rotulo = rotulo; this.indice = indice; }
  public void setPosicionEnVertices( Position<Vertex<V>> p ) {
          posicionEnVertices = p; }
  public void setRotulo(V nuevoRotulo) { rotulo = nuevoRotulo; }
  public int getIndice() { return indice; }
  public Position<Vertex<V>> getPositionEnVertices() { return posicionEnVertices; }
  public V element() { return rotulo; }
}
```



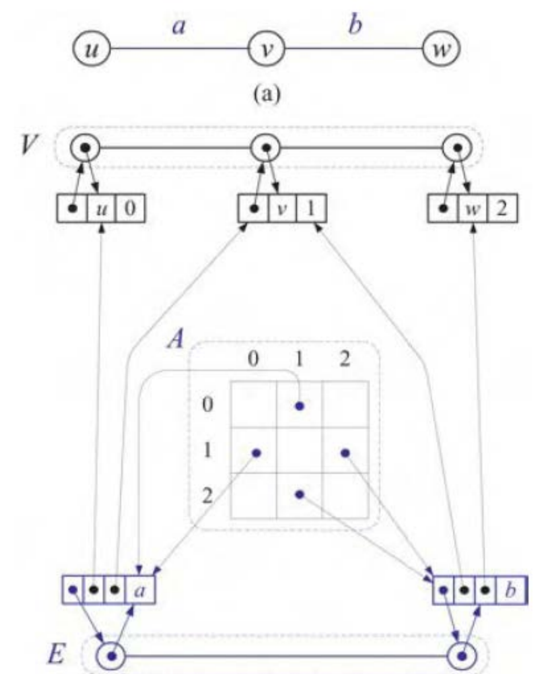
```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

private class Arco<V,E> implements Edge<E> {
private E rotulo;
private Position<Edge<E>> posicionEnArcos;
private Vertice<V> v1, v2;

public Arco( E rotulo, Vertice<V> v1, Vertice<V> v2 ) {
            this.rotulo = rotulo; this.v1 = v1;  this.v2 = v2; }
public void setPosicionEnArcos( Position<Edge<E>> p ) { posicionEnArcos = p; }
public E element() { return  rotulo; }
public Position<Edge<E>> getPosicionEnArcos() { return posicionEnArcos; }
public Vertice<V> getV1() { return v1; }
public Vertice<V> getV2() { return v2; }
public void setRotulo(E nuevoRotulo) { rotulo = nuevoRotulo; }
}
```



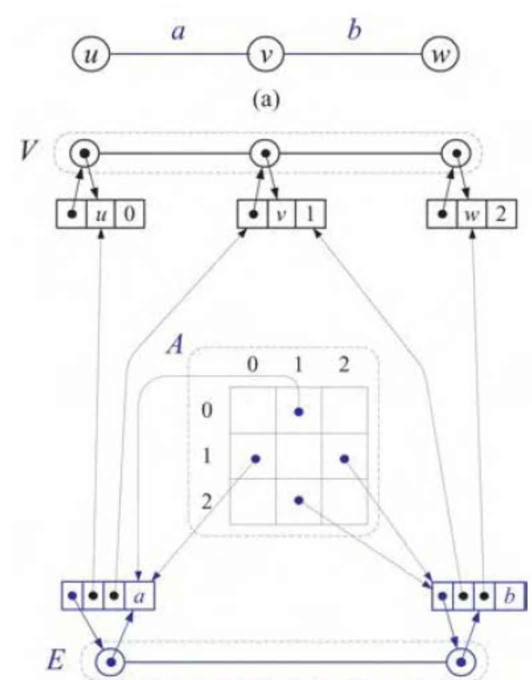
```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

public GrafoConMatriz( int n ){   // Recibe el tamaño de la matriz
            vertices = new DoubleLinkedList<Vertex<V>>();
            arcos = new DoubleLinkedList<Edge<E>>();
            // Produce warning
            // unsafe operation: compilar con javac -Xlint:unchecked
            // GrafoConMatriz.java
            matriz = (Edge<E> [][]) new Arco[n][n];
            cantidadVertices = 0;

            // Innecesario en Java:
            for(int i=0; i<n; i++ )
                        for(int j=0; j<n; j++ )
                                    matriz[i][j] = null;
}
```



```
//		protected PositionList<Vertex<V>> vertices;
//		protected PositionList<Edge<E>> arcos;
//		protected Edge<E> [][] matriz;
//		protected int cantidadVertices;

public Vertex<V> insertVertex(V x)  {
		Vertice<V> vv = new Vertice<V>(x, cantidadVertices++);
		vertices.addLast( vv );
		vv.setPosicionEnVertices( vertices.last() );
		return vv;
}
```

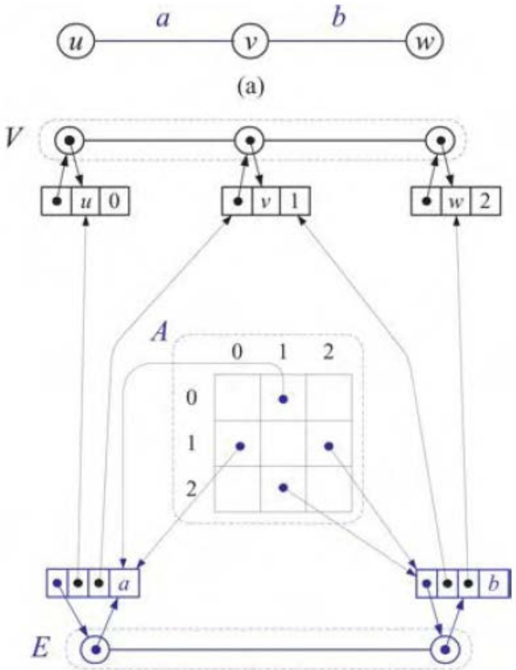


```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E x)  {
        // Cargo arco en la matriz:
        Vertice<V> vv = (Vertice<V>) v;
        Vertice<V> ww = (Vertice<V>) w;
        int fila = vv.getIndice();
        int col = ww.getIndice();
        Arco<V,E> arco = new Arco<V,E>( x, vv, ww );
        // Grafo no dirigido => matriz simétrica

        matriz[fila][col] = matriz[col][fila] = arco;

        // Cargo arco en la lista de arcos:
        arcos.addLast( arco );
        arco.setPosicionEnArcos( arcos.last() );
        return arco;
}
```



```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

public Iterable<Vertex<V>> vertices() {
            PositionList<Vertex<V>> lista = new DoubleLinkedList<Vertex<V>>();
            for( Vertex<V> v : vertices )
                        lista.addLast(v);
            return lista;
}

public Iterable<Edge<E>> edges() {
            PositionList<Edge<E>> lista = new DoubleLinkedList<Edge<E>>();
            for( Edge<E> e : arcos )
                        lista.addLast(e);
            return lista;
}
```

```java
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

public Iterable<Edge<E>> incidentEdges(Vertex<V> v) {
            Vertice<V> vv = (Vertice<V>) v;
            int fila = vv.getIndice();
            PositionList<Edge<E>> lista = new DoubleLinkedList<Edge<E>>();
            for( int col = 0; col < cantidadVertices; col++ )
                        if( matriz[fila][col] != null )
                                    lista.addLast( matriz[fila][col] );
            return lista;
}
```



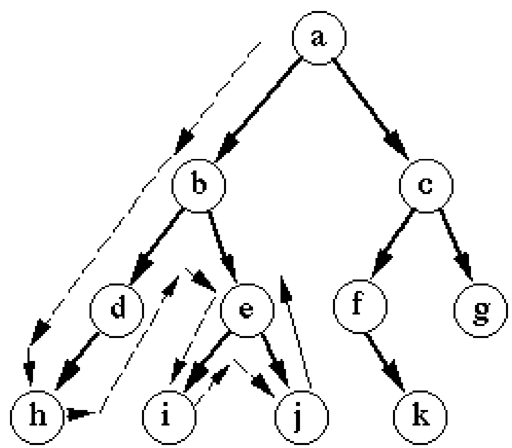
```
//        protected PositionList<Vertex<V>> vertices;
//        protected PositionList<Edge<E>> arcos;
//        protected Edge<E> [][] matriz;
//        protected int cantidadVertices;


public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
        throws GraphException {
        Arco<V,E> ee = (Arco<V,E>) e;
        if( ee.getV1() == v ) return ee.getV2();
        else if( ee.getV2() == v ) return ee.getV1();
        else throw new
                GraphException( "Vertice y arco no relacionados");
}
```

```
//          protected PositionList<Vertex<V>> vertices;
//          protected PositionList<Edge<E>> arcos;
//          protected Edge<E> [][] matriz;
//          protected int cantidadVertices;

public Vertex<V> [] endVertices(Edge<E> e) {
            Vertex<V> [] a = (Vertex<V> []) new Vertice[2];
            Arco<V,E> ee = (Arco<V,E>) e;
            a[0] = ee.getV1();
            a[1] = ee.getV2();
            return a;
}

public boolean areAdjacent(Vertex<V> v, Vertex<V> w)  {
            Vertice<V> vv = (Vertice<V>) v;
            Vertice<V> ww = (Vertice<V>) w;
            int i = vv.getIndice();
            int j = ww.getIndice();
            return matriz[i][j] != null;
}
```

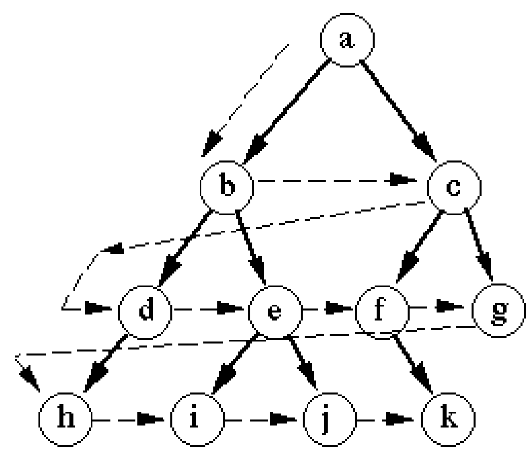
```
public E removeEdge(Edge<E> e)  {
        try {
                Arco<V,E> ee = (Arco<V,E>) e;
                int fila = ee.getV1().getIndice();
                int col = ee.getV2().getIndice();
                matriz[fila][col] = matriz[col][fila] = null;
                arcos.remove( ee.getPosicionEnArcos() );
                return e.element();
        } catch( InvalidPositionException exc ) {
            System.out.println( "GrafoConMatriz:removeEdge: Error?");
            return null;
        }
}
```

Recorridos de grafos

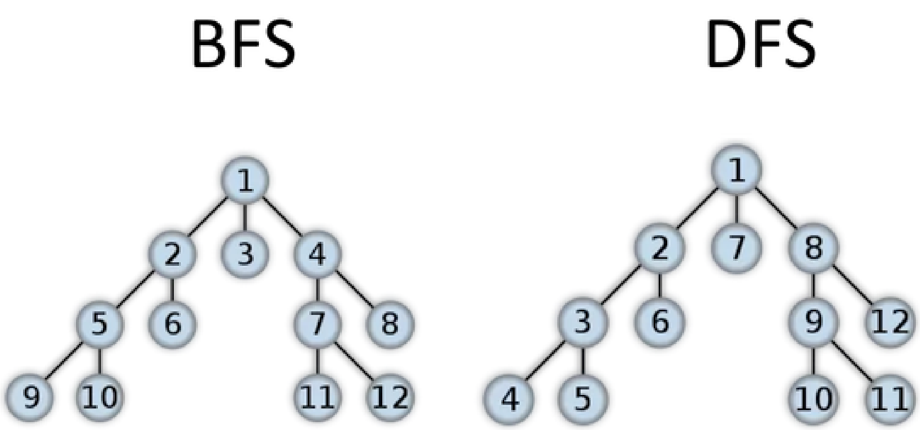
- <u>En profundidad (Depth-First Search o DFS):</u> (Equivale al recorrido pre o post orden en árboles + un testeo para no volver a recorrer un subgrafo ya explorado): a, b, d, h, e, i, j, c, f, k, g
- <u>En anchura (Breadth-First Search o BFS):</u> (Equivale al recorrido por niveles en árboles + un testeo para no volver a recorrer un subgrafo ya explorado): a, b, c, d, e, f, g, h,i, j, k



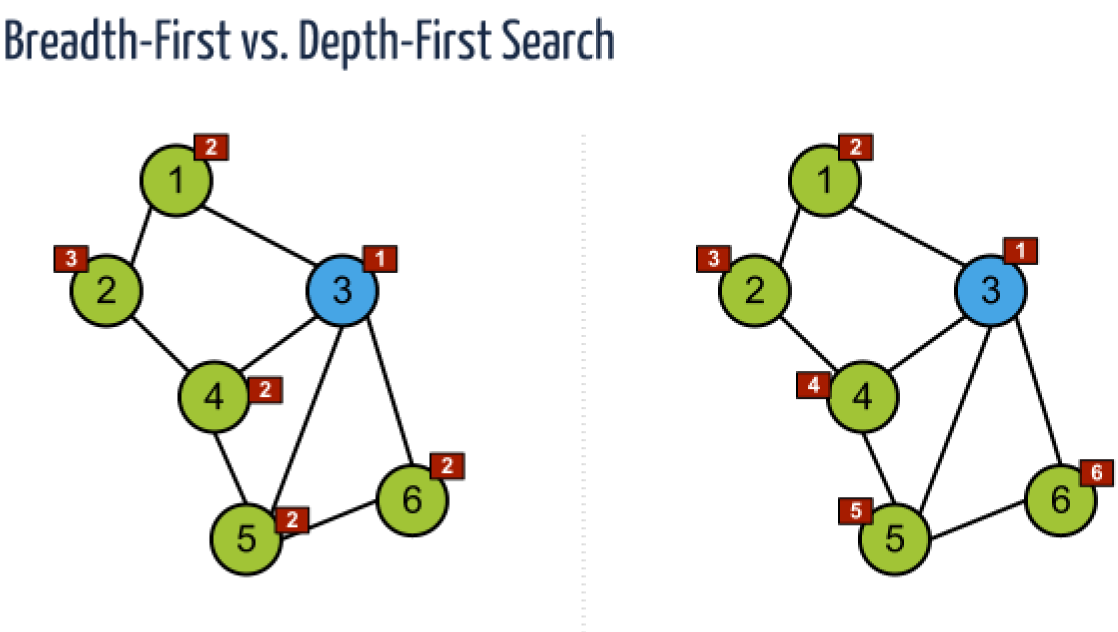
Depth-first search



Breadth-first search



El número del vértice corresponde al orden en el cual es visitado por el algoritmo respectivo.



Ambos recorridos salen del 3. En rojo, está la distancia al 3 en cantidad de arcos de acuerdo al recorrido.

BFS: 3, 1, 4, 5, 6, 2

DFS: 3, 1, 2, 4, 5, 6

(Imágenes tomadas de la web)

Búsqueda en profundidad

- Una búsqueda en profundidad (DFS o Depth-First Search) permite recorrer todos los vértices de un grafo de manera ordenada.
- Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto.
- Cuando ya no quedan más nodos que visitar en dicho camino, regresa (backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Búsqueda en profundidad

```
Algoritmo DFSShell( G : Grafo )
    para cada vértice v de G hacer
        marcar v como no visitado
    para cada vértice v de G hacer
        si v no está visitado entonces
            DFS( G, v )

Algoritmo DFS( G : Grafo; v : Vértice )
        procesamiento de v previo al recorrido
        marcar a v como visitado
        para cada vértice w adyacente a v en G hacer
            si w no está visitado entonces
                DFS( G, w )
        procesamiento de v posterior al recorrido
```

$T_{DFS}(n,m) = O(n+m)$

Opciones para implementar las marcas de visitado

1. Usar un mapeo externo al grafo de Vertex<V> en Boolean
   - Ventaja: No hay que modificar el grafo que ya programamos.
   - Contra: El tiempo de marcar y desmarcar puede tender a O(n)
2. Agregar un boolean a la clase Vertice del grafo
   - Contra: Por cada algoritmo que escribo tengo que ensuciar el grafo agregando atributos y operaciones
   - Ventaja: Puedo garantizar que marcar y desmarcar funciona en O(1)
3. Decorar los vértices del grafo (opción de GT).
   - Ventaja: En forma abstracta mantengo toda la información del DFS y de futuros algoritmos
   - Contra: Hay que modificar lo que ya programamos

Opción 3: DFS con vértices decorados

Una posición decorada es una posición que además es un mapeo.

public interface DecorablePosition<E>

extends Position<E>, Map<Object,Object> { }

public interface Vertex<E> extends DecorablePosition<E> { }

Para implementar la clase Nodo, hago:

```
public class Nodo<V,E>
        implements Vertex<V>
        extends HashMap<Object,Object> {
                … Idem a lo presentado en las clases anteriores …
        }
```

Vértices decorados

Si v es un Vertex<Integer>, entonces:

- v.element() retorna un entero que corresponde al rótulo de v
- v.put(ESTADO, VISITADO) permite anotar que v está visitado
- v.get(ESTADO) permite testear si v es un vértice visitado o no (que puede devolver NO_VISITADO o null).

Opción 3: DFS con vértices decorados

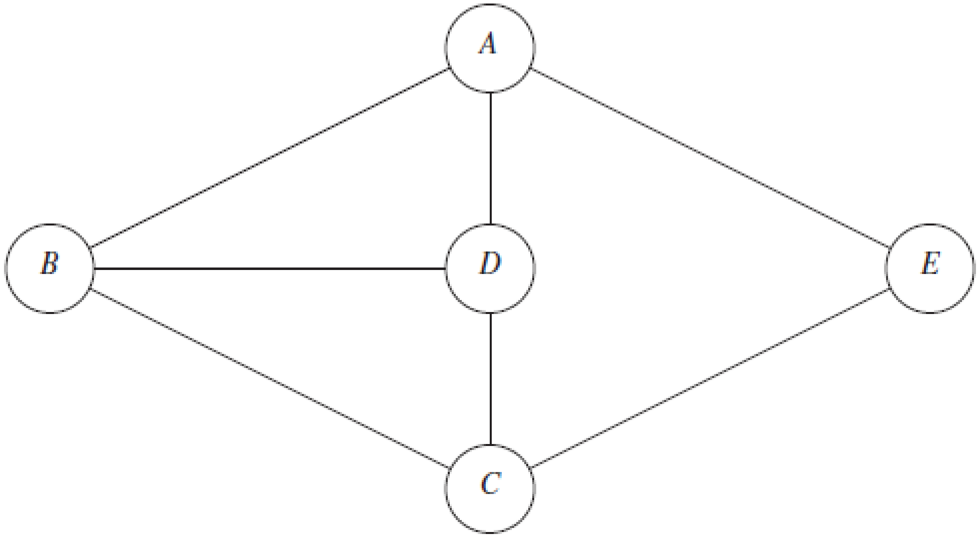
```
public class Aplicación {
        private final Object VISITADO = new Object();
        private final Object NOVISITADO = new Object() ;
        private final Object ESTADO = new Object();

        public <V,E> static void dfsShell( Graph<V,E> g ) {
                for( Vertex<V> v : g.vertices() )
                        v.put( ESTADO, NOVISITADO );
                for( Vertex<V> v : g.vertices() )
                        if( v.get( ESTADO ) == NOVISITADO )
                                dfs( g, v );
        }
        private <V,E> static void dfs( Graph<V,E> g, Vertex<V> v ) { …. }
}
```

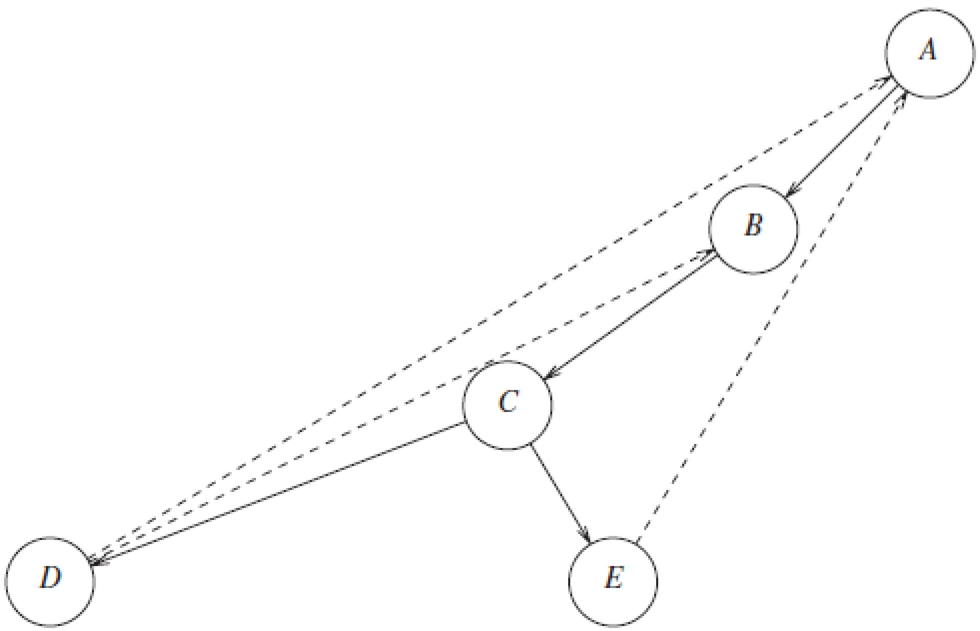
Opción 3: DFS con vértices decorados

```
private <V,E> static void dfs( Graph<V,E> g, Vertex<V> v ) {

        // El procesamiento de v es sólo imprimir su rótulo
        System.out.println( v.element() );

        v.put( ESTADO, VISITADO );
        Iterable<Edge<E>> adyacentes = g.emergentEdges( v );
        for( Edge<E> e : adyacentes ) {
                Vertex<V> w = g.opposite( v, e );
                if( w.get( ESTADO ) == NOVISITADO )
                        dfs( g, w );
        }

        // Acá va el postprocesamiento de v
}
```

Bosque del DFS en grafos no dirigidos

- Se asume que el grafo es conexo (se puede testear haciendo un DFS y viendo que se visitan todos los vértices).
- Al orientar los arcos en la dirección en la que son explorados durante el recorrido, se distinguen:
  - <u>Arcos de descubrimiento o árbol</u> (discovery o tree edges): Arcos que llevan a vértices no visitados
  - <u>Arcos de retroceso (back edges):</u> Arcos que llevan a vértices que ya fueron visitados.





Para este grafo no dirigido, debajo está el árbol que se genera al salir de A (los arcos llenos son los arcos tree y los árcos punteados son los back).

Si el grafo es no conexo, tendremos un árbol por cada llamada al DFS (i.e. para cada componente conexa). Gráfico tomado de *Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, Third Edition, Pearson, 2012.*

Bosque del DFS en grafos no dirigidos

Algoritmo DFS(G : Grafo; v : Vertice)
Marcar a v como visitado
Para cada arco e en G.incidentEdges(v) hacer
    si e no está visitado entonces
        w ← G.opposite(v, e)
        si w no está visitado entonces
            etiquetar a e como *arco de descubrimiento*
            DFS(G, w)
        sino
            etiquetar a e como *arco de retroceso*

$T_{DFS}(n,m) = O(n+m)$ con lista de adyacencia
$T_{DFS}(n,m) = O(n^2)$ con matriz de adyacencia

Bosque del DFS con arcos decorados

```
public static void <V,E> DFS( Graph<V,E> G, Vertex<V> v, Object
k ) {
v.put( k, VISITADO );
for( Edge<E> e : G.incidentEdges(v) ) {
    if( e.get( k ) == null ) {
        w = G.opposite(v, e )
        if( w.get( k ) == null ) {
            e.put( k, ARCO_DESCUBRIMIENTO );
            DFS( G, w, k );
        } else
            e.put( k, ARCO_RETROCESO );
}
```

$T_{DFS}(n,m) = O(n+m)$ asumiendo operaciones de mapeo en $O(1)$

Aplicaciones del DFS para grafos no dirigidos en O(n+m)

- Testear si G es conexo (todos los vértices quedan visitados si y sólo si el grafo es conexo)
- Calcular un árbol abarcador si G es conexo (formado por los vértices de G y por sus arcos tree)
- Calcular las componentes conexas (por cada iteración de DFSShell incremento un contador indicando el número de componente conexa y con ese contador etiqueto los vértices de cada componente)
- Encontrar un camino entre dos nodos (clase siguiente)
- Encontrar un ciclo (clase siguiente)

Búsqueda en anchura (BFS)

- La búsqueda en anchura (BFS o Breadth First Search) es un algoritmo para recorrer o buscar elementos en un grafo.
- Se comienza eligiendo algún nodo como elemento raíz y se exploran todos los vecinos de este nodo.
- A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

Búsqueda en anchura (Breadth-First Search o BFS)

```
Algoritmo BFSShell( G : Grafo )
    para cada vértice v de G hacer
        marcar v como no visitado
    para cada vértice v de G hacer
        si v no está visitado entonces
            BFS( G, v )

Algoritmo BFS( G: Grafo; v : Vertice)
        …. En próxima diapositiva …
```

Búsqueda en anchura

```
Algoritmo BFS( G : Grafo; v : Vértice )
        cola ← new Cola()
        cola.enqueue( v )
        mientras not cola.isEmpty() hacer
                w ← cola.dequeue()
                procesar a w
                para cada vértice x adyacente a w hacer
                        si x no está visitado entonces
                                marcar a w como visitado
                                cola.enqueue( x )
```

$T_{BFS}(n,m) = O(n+m)$ (justificación en la próxima diapositiva)

Análisis del tiempo de ejecución

- Sea un grafo G=(V,A)
- Sean n = #V y m = #A (#S quiere decir el cardinal de S)
- Para simplificar el análisis, supongamos que el grafo es conexo.
- Sea A_i la cantidad de adyacentes del vértice i:

$$
\begin{aligned}
T(n,m) &= c_1 + \sum_{i=1}^{n}\left(c_2 + \sum_{j=1}^{A_i} c_3\right) \\
&= c_1 + \sum_{i=1}^{n}(c_2 + A_i c_3) \\
&= c_1 + \sum_{i=1}^{n} c_2 + \sum_{i=1}^{n} A_i c_3 \\
&= c_1 + nc_2 + mc_3 = O(n+m)
\end{aligned}
$$

Note que $m = O(n^2)$

Aplicaciones del BFS

Idem DFS y además:

- Hallar el camino más corto (en cantidad de arcos) entre dos vértices (en O(n+m)) (en clase siguiente).

Bibliografía

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