Agenda - Grafos

- Conceptos
- Representaciones: matriz de adyacencias y lista de adyacencias
- Implementaciones en Java
- Recorridos
  - En profundidad: **DFS** (Depth First Search)

    Ejemplo: Encontrar los vuelos con un costo determinado
  - En amplitud: **BFS** (Breadth First Search)

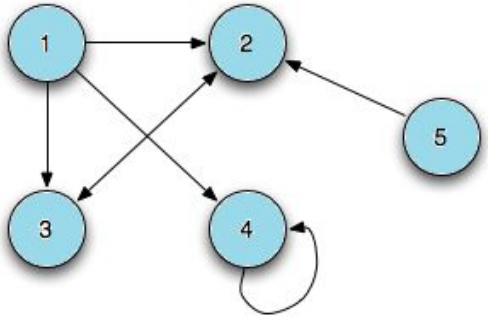
    Ejemplo: Tiempo de infección de una red

Grafos en JAVA
Conceptos

Un **grafo** es un **conjunto de nodos relacionados**. Más especificamente, un grafo es par ordenado de **conjuntos finitos G=(V,A)** donde **V** es el **conjunto** finito de **vértices** y **A** es el **conjunto** finito de **Aristas**.

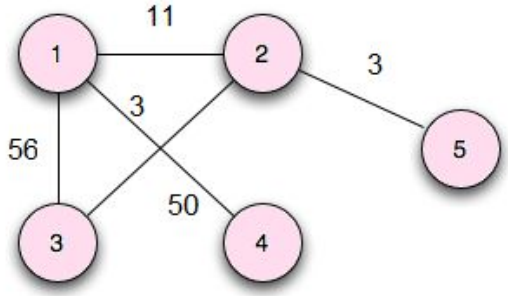
En general, los **vértices** son **nodos** de **procesamiento** o estructuras que contienen algún tipo de **información** mientras que las **aristas** son las **relaciones** entre los vértices. Las **artistas** también pueden contener **información** (distancia, peso, etc.) en cuyo caso es un **grafo pesado**.

En cualquiera de los dos grafos, las relaciones las determinan las **aristas**.



Si se utilizan flechas para conectar los nodos decimos que el **grafo es dirigido** (también llamado **digrafo**).

Grafo no pesado



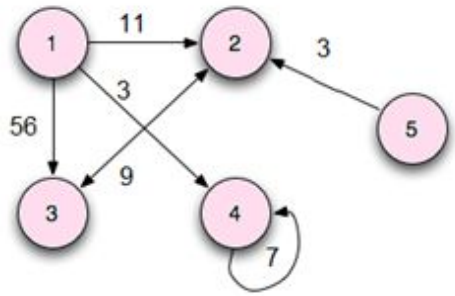
Si la conexión entre los vértices no tiene dirección, el **grafo es no dirigido**.

Grafo pesado

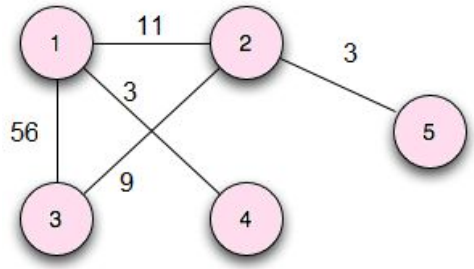
Grafos en JAVA
Representaciones

Las representaciones más comunes de grafos son **matriz de adyacencias** y **lista de adyacencias**.

- **Matriz de adyacencias**: el grafo se representa como una matriz de |V| x |V|, con valores enteros (o de otro tipo de dato).



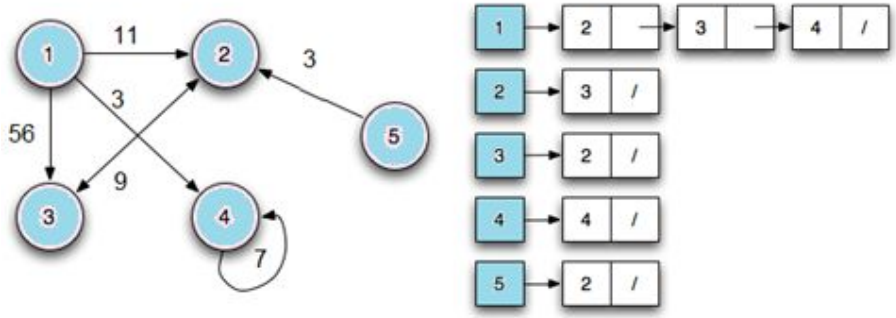
	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0

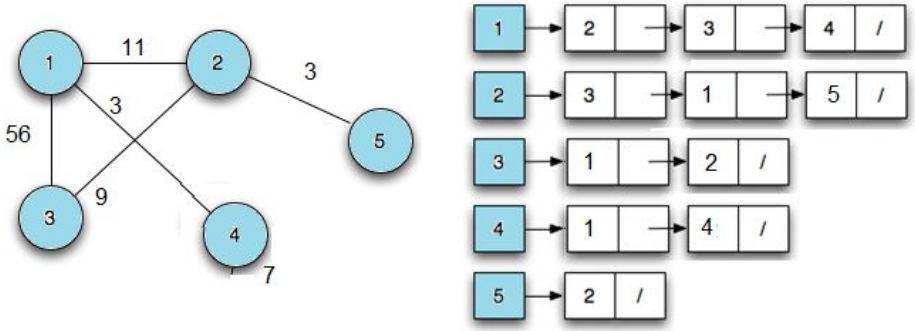


Matriz simétrica

	1	2	3	4	5
1	0	11	56	3	0
2	11	0	9	0	3
3	56	9	0	0	0
4	3	0	0	0	0
5	0	3	0	0	0

- **Lista de adyacencias**: el grafo G=(V,A) se representa como un arreglo/lista de |V| de vértices.

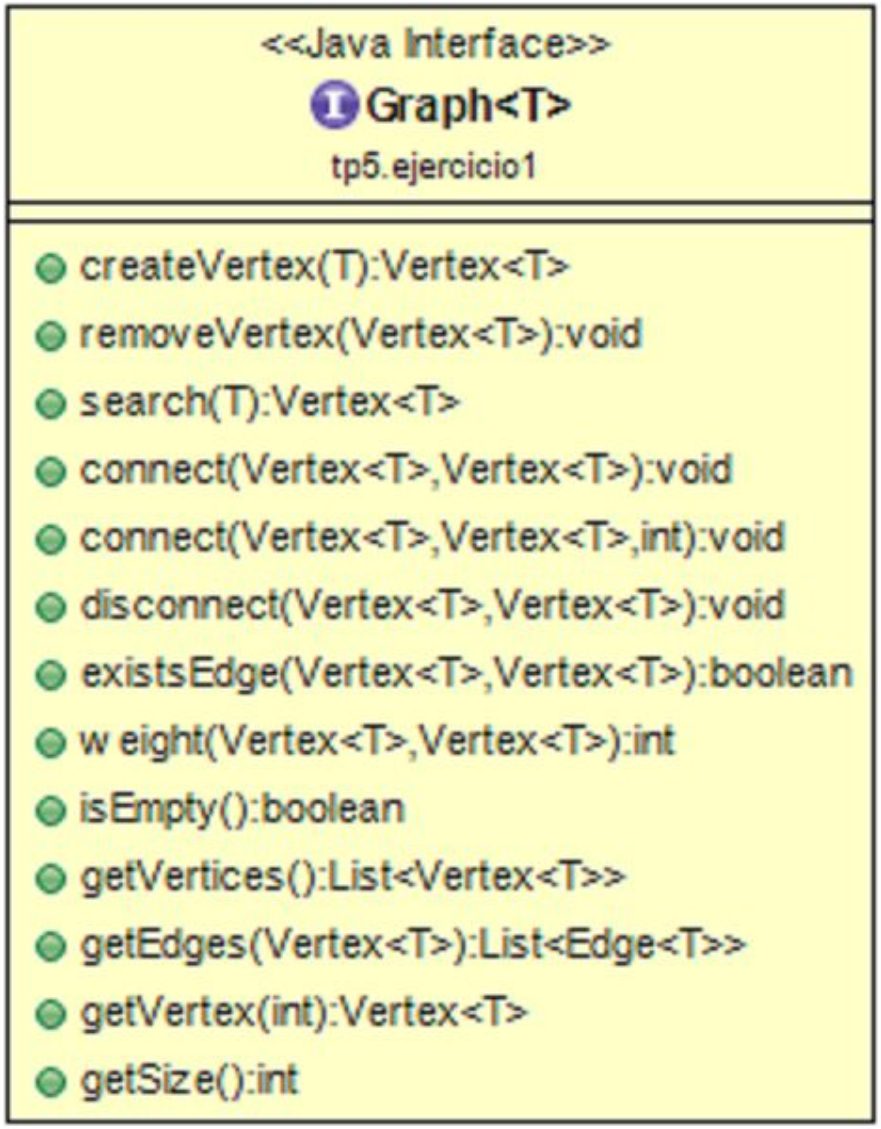


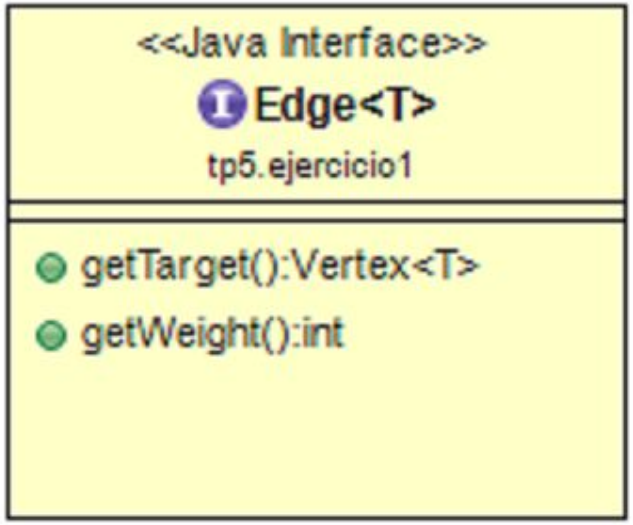


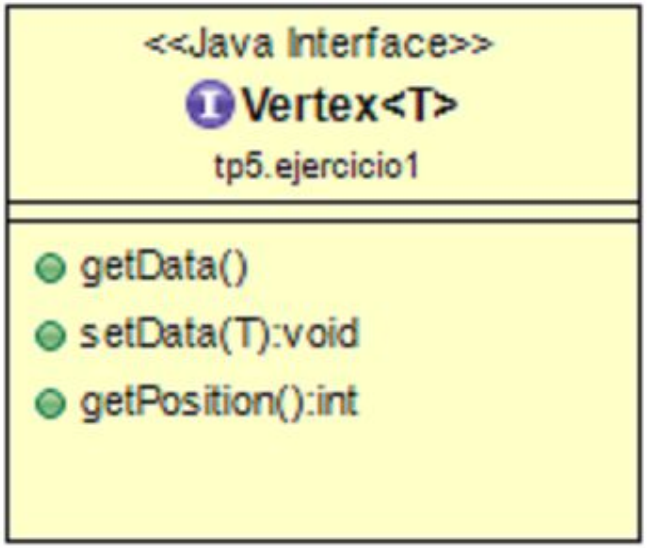
Grafos en JAVA
Interfaces para definir Grafos

¿Por qué se usan interfaces?

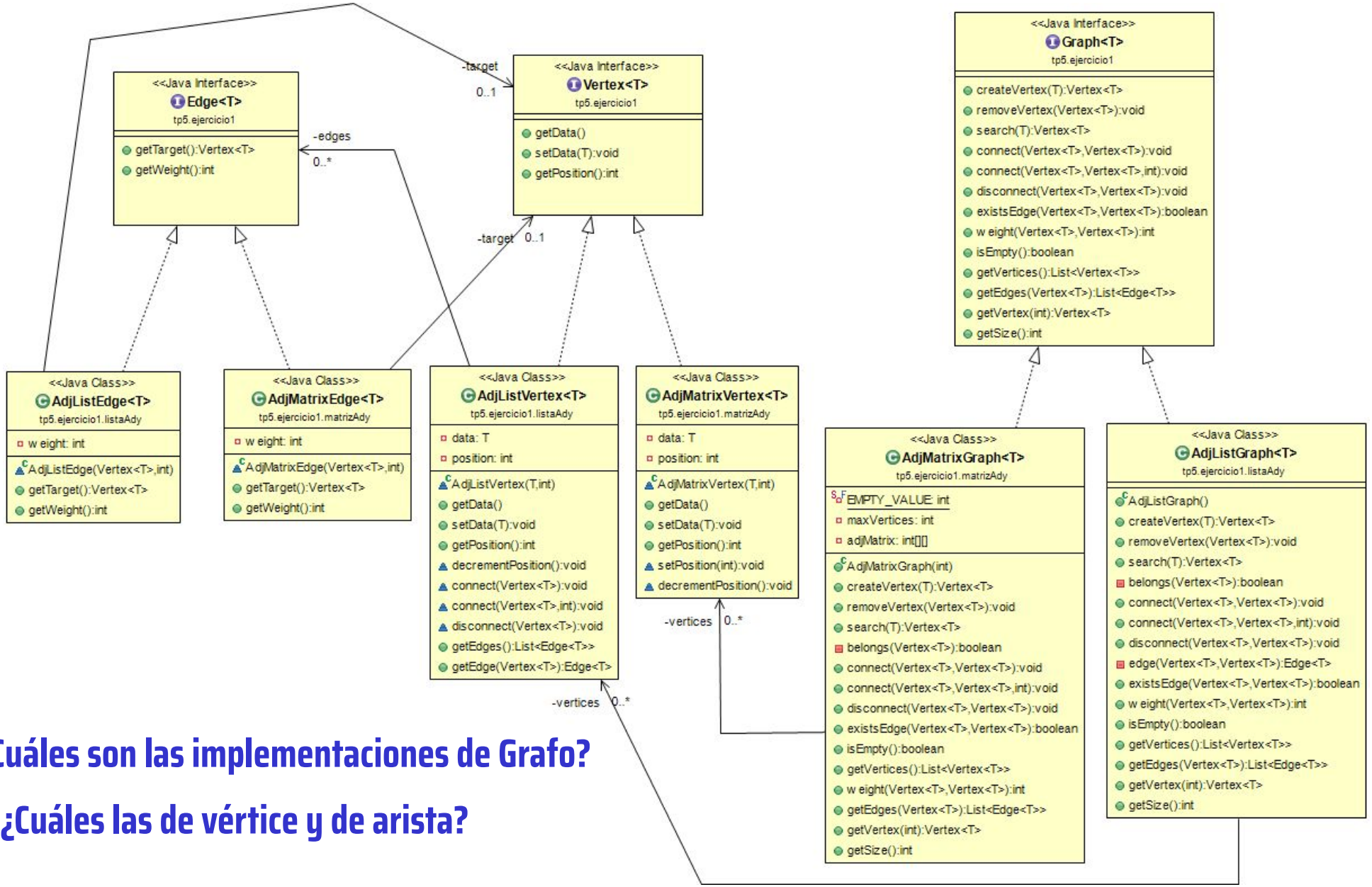
¿Cómo está representado el peso?







Grafos - La interfaces y clases



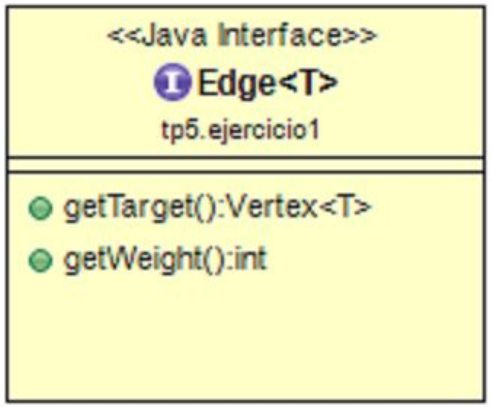
¿Cuáles son las implementaciones de Grafo?

y ¿Cuáles las de vértice y de arista?

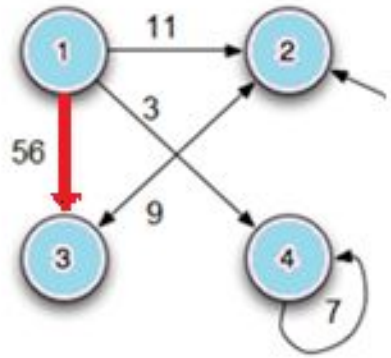
Grafos con Lista de Adyacencias

Clase que implementa la interface **Edge** (arista)

```
package package tp5.ejercicio1.adjList;

public class AdjListEdge<T> implements Edge<T> {
    private Vertex<T> target;
    private int weight;

    AdjListEdge(Vertex<T> target, int weight){
        this.target = target;
        this.weight = weight;
    }

    @Override
    public Vertex<T> getTarget() {
        return target;
    }

    @Override
    public int getWeight() {
        return weight;
    }
}
```



Una **arista** siempre tiene un **destino** (target) y podría tener un **peso**



¿Qué objeto crea objetos AdjListEdge (aristas)?

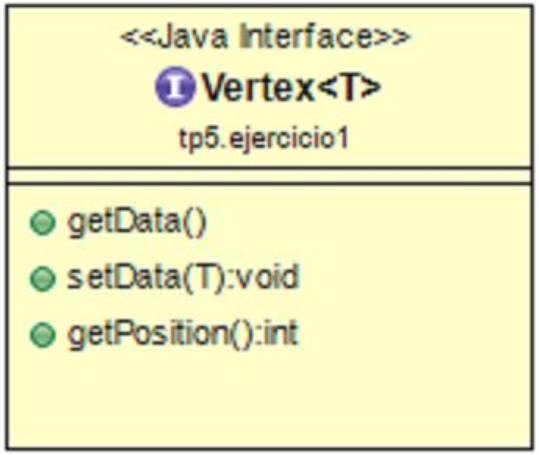
Grafos con Lista de Adyacencias

Clase que implementa la interface **`Vertex`** (vértice)

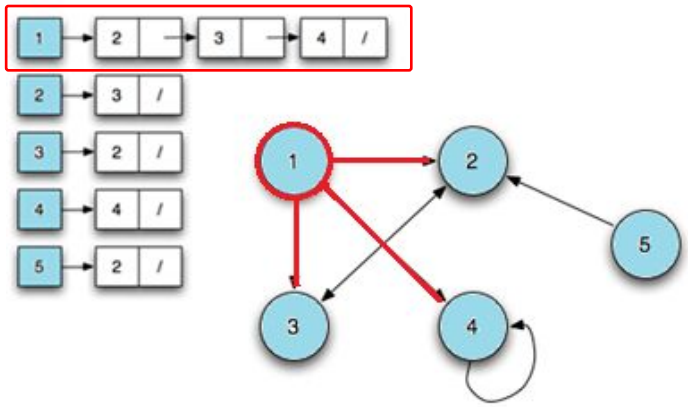
```
public class AdjListVertex<T> implements Vertex<T> {
      private T data;
      private int position;
      private List<Edge<T>> edges;
      AdjListVertex(T data, int position) {
       this.data = data;
       this.position = position;
       this.edges = new ArrayList<>();
      }
      @Override
      public T getData() { return this.data; }
      @Override
      public void setData(T data) {this.data = data; }
      @Override
      public int getPosition() {
        return this.position;
      }
      public List<Edge<T>> getEdges() {
        return this.edges;
      }
      public Edge<T> getEdge(Vertex<T> destination) {
         for (Edge<T> edge : this.edges) {
           if (edge.getTarget() == destination)
                return edge;
         }
       return null;
      }
     void connect(Vertex<T> destination, int weight) {
       Edge<T> edge = this.getEdge(destination);
       if (edge == null)
           // se crea solo si no existe
            this.edges.add(new AdjListEdge<>(destination, weight));
     }
}
```

¿Para qué se utiliza el método getPosition()?

¿Qué objeto crea objetos AdjListVertex (vértices)?



Un **vértice** contiene un **dato** y **tiene** una **lista de adyacentes**. La **lista de adyacentes** es una lista de **aristas** y cada una **tiene** un **vértice destino**.

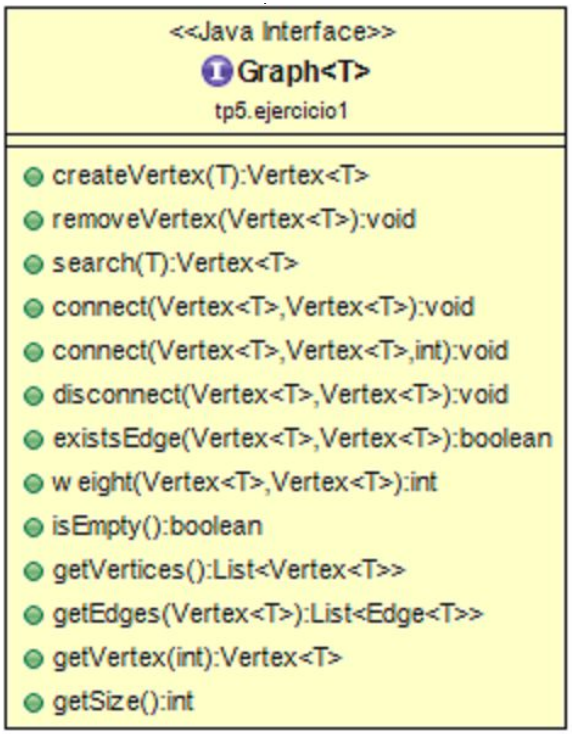


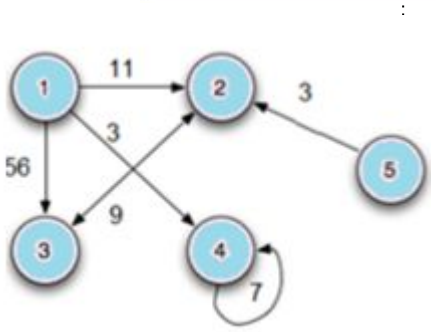
Profesores: Laura Fava - Pablo Iuliano

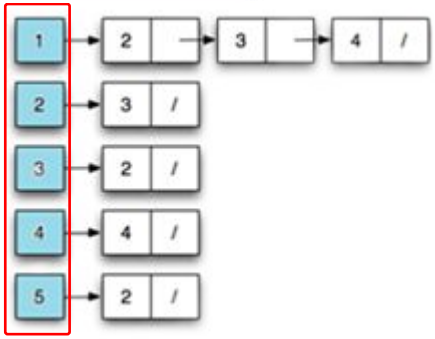
Grafos con Lista de Adyacencias

Clase que implementa la interface `Graph`

```
public class AdjListGraph<T> implements Graph<T> {

      private List<AdjListVertex<T>> vertices;

      public AdjListGraph() {
            this.vertices = new ArrayList<>();
      }
      @Override
      public Vertex<T> createVertex(T data) {
        int newPos = this.vertices.size();
        AdjListVertex<T> vertex = new AdjListVertex<>(data, newPos);
        this.vertices.add(vertex);
        return vertex;
      }
      @Override
      public void removeVertex(Vertex<T> vertex) {
        int position = vertex.getPosition();
        if (this.vertices.get(position) != vertex)
             // el vértice no corresponde al grafo
             return;
        this.vertices.remove(position);
        for (;position < this.vertices.size(); position++)
             this.vertices.get(position).decrementPosition();
        for (Vertex<T> other : this.vertices)
                  this.disconnect(other, vertex);
       }
       @Override
       public Vertex<T> search(T data) {
         for (Vertex<T> vertex : this.vertices) {
             if (vertex.getData().equals(data))
                  return vertex;
       return null;
       }
}
```







¿Funciona el search() para Grafos de Integer? y ¿de String? y ¿de objetos Persona?

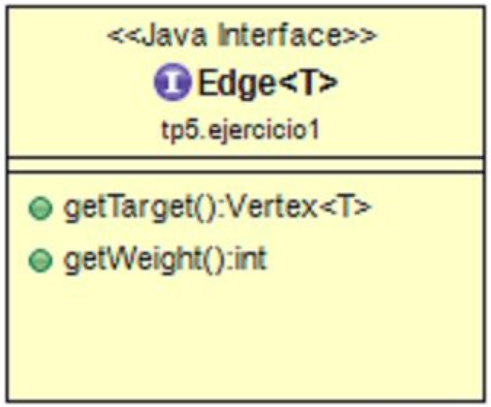
¿Cómo aseguramos que funcione para Persona?

Un **Grafo** está formado por una **lista de vértices** y cada **vértice** contiene una **lista de aristas** que representa a la lista de **vértices adyacentes**

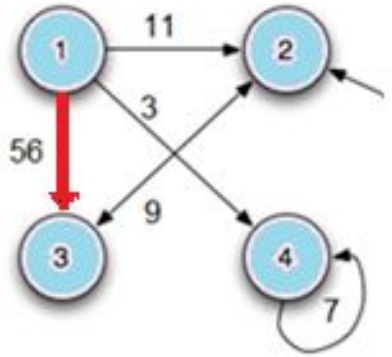
Grafos con Matriz de Adyacencias

Clase que implementa la interface **Edge** (arista)

```
package tp5.ejercicio1.adjMatrix;

public class AdjMatrixEdge<T> implements Edge<T> {
    private Vertex<T> target;
    private int weight;

    AdjMatrixEdge(Vertex<T> target, int weight){
        this.target = target;
        this.weight = weight;
    }

    @Override
    public Vertex<T> getTarget() {
        return target;
    }

    @Override
    public int getWeight() {
        return weight;
    }
}
```



Una **arista** siempre **tiene** el **destino** y podría tener un peso



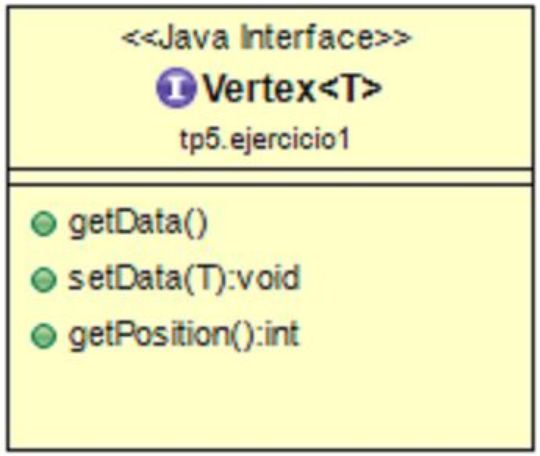
¿Qué objeto crea objetos AdjMatrixEdge (aristas)?

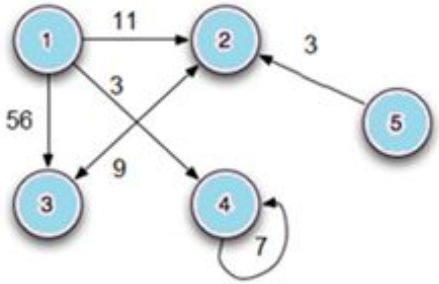
Grafos con Matriz de Adyacencias

Clase que implementa la interface `Vertex`

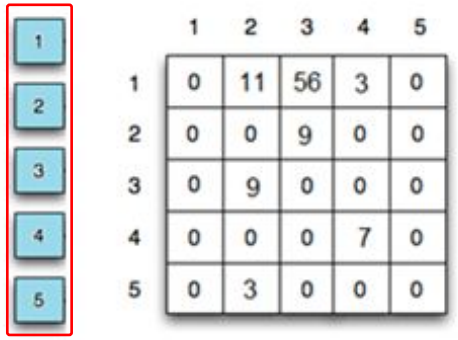
```
package tp5.ejercicio1.adjMatrix;

public class AdjMatrixVertex<T> implements Vertex<T> {
	private T data;
	private int position;

	AdjMatrixVertex(T data, int position) {
		this.data = data;
		this.position = position;
	}

	@Override
	public T getData() {
		return this.data;
	}
	@Override
	public void setData(T data) {
		this.data = data;
	}
	@Override
	public int getPosition() {
		return position;
	}

	void setPosition(int position) {
		this.position = position;
	}

	void decrementPosition() {
		this.position--;
	}
}
```

¿Qué objeto crea objetos AdjMatrixVertex (vértices)?





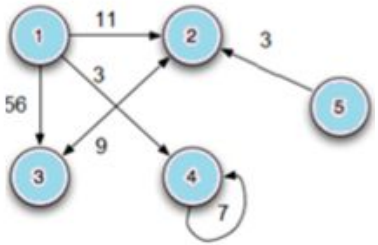
Un **vértice tiene** un **dato** y una **posición**

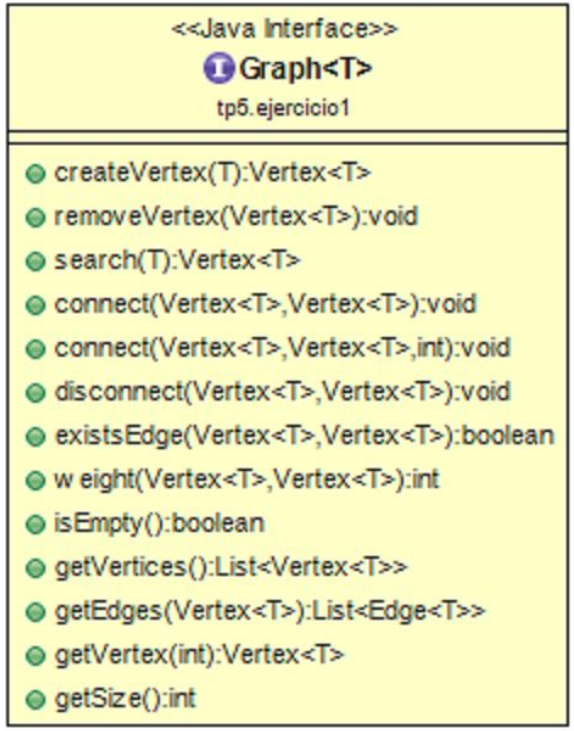


Grafos con Matriz de Adyacencias

Clase que implementa la interface `Graph`

```
package tp5.ejercicio1.adjMatrix;

public class AdjMatrixGraph<T> implements Graph<T> {
    private static final int EMPTY_VALUE = 0;
    private int maxVertices;
    private List<AdjMatrixVertex<T>> vertices;
    private int[][] adjMatrix;

    public AdjMatrixGraph(int maxVert) {
        maxVertices = maxVert;
        vertices = new ArrayList<>();
        adjMatrix = new int[maxVertices][maxVertices];
    }
    @Override
    public Vertex<T> createVertex(T data) {
      if (vertices.size() == maxVertices)
            // se llego al máximo
            return null;
      AdjMatrixVertex<T> vertice = new AdjMatrixVertex<>(data, vertices.size());
      vertices.add(vertice);
      return vertice;
    }
    @Override
    public List<Edge<T>> getEdges(Vertex<T> v) {
        List<Edge<T>> ady = new ArrayList<Edge<T>>();
        int veticePos = v.getPosition();
        for (int i = 0; i < vertices.size(); i++) {
           if (adjMatrix[veticePos][i] != EMPTY_VALUE) {
             ady.add(new AdjMatrixEdge<T>(vertices.get(i), adjMatrix[veticePos][i]));
        }
        return ady;
    }
}
```





	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0

¿Qué diferencias observan en las implementaciones de getEdges() con LA y con MA?

Un **`Grafo`** tiene una **lista de `vértices`** y la matriz representa la **lista de aristas**

Grafos
DFS (Depth First Search)

El DFS es un algoritmo de recorrido de grafos en profundidad. Es una generalización del recorrido preorden de un árbol.

Esquema recursivo

Dado G = (V , A)

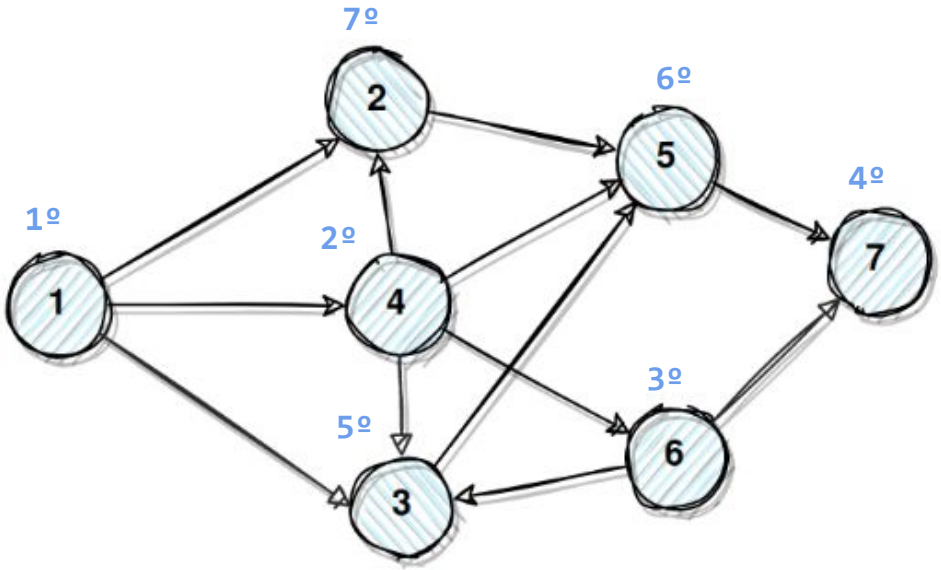
1. Marcar todos los vértices como no visitados.
2. Elegir vértice u (no visitado) como punto de partida.
3. Marcar u como visitado.
4. Para todo v adyacente a u, (u,v) Є A, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

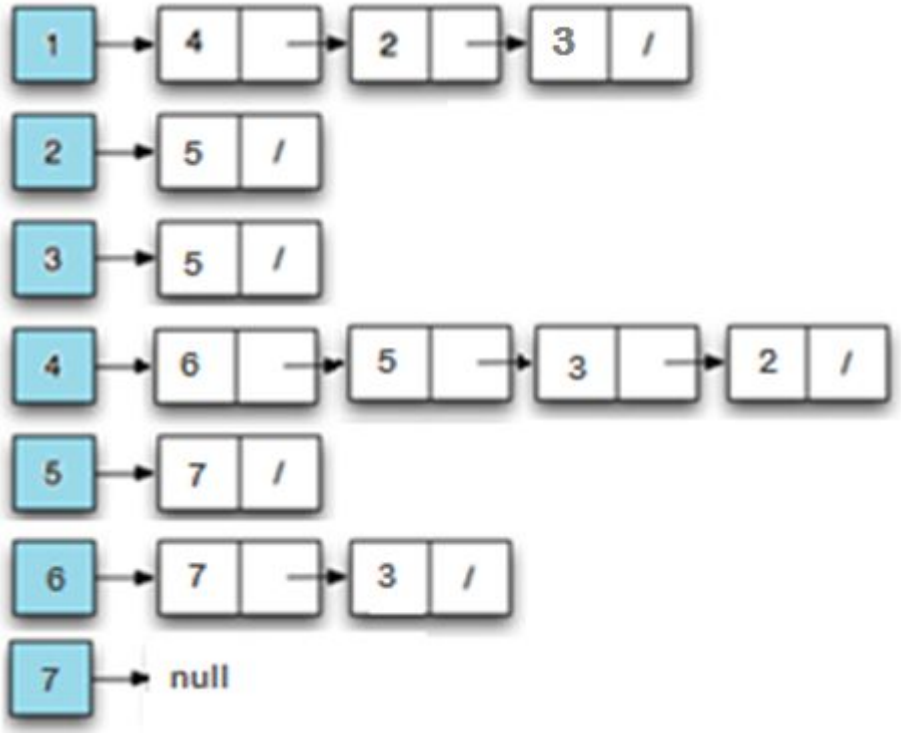
¿Cuándo finaliza el recorrido?

- Cuando se visitaron todos los nodos alcanzables desde u.
- Si desde u no fueron alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

Grafos
DFS (Depth First Search)

El recorrido del DFS depende del orden en que aparecen los vértices en las listas de adyacencia.





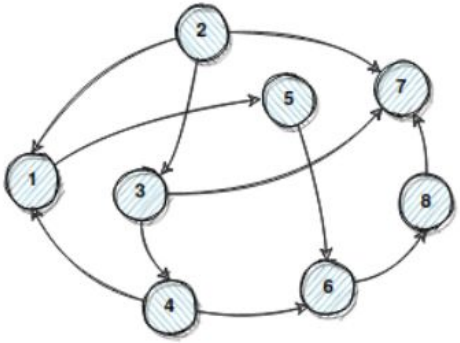
Tomamos como punto de partida el vértice 1, ¿cuál es un posible DFS?

Grafos
DFS (Depth First Search)

```
public class Recorridos<T> {
  public void dfs(Graph<T> grafo) {
     boolean[] marca = new boolean[grafo.getSize()];
     for (int i = 0; i < grafo.getSize(); i++) {
       if (!marca[i]) {
         System.out.println("largo con: "+grafo.getVertex(i).getData());
         dfs(i, grafo, marca);
       }
     }
   }
   private  void dfs(int i, Graph<T> grafo, boolean[] marca) {
     marca[i] = true;
     Vertex<T> v = grafo.getVertex(i);
     System.out.println(v.getData());
     List<Edge<T>> adyacentes = grafo.getEdges(v);  //adyacentes de V
     for (Edge<T> e: adyacentes){
          int j = e.getTarget().getPosition();
          if (!marca[j]) {
               dfs(j, grafo, marca);
          }
     }
   }
}
```

equivalente

```
Iterator<Edge<T>> it = adyacentes.iterator();
while (it.hasNext()) {
     int j = it.next().getTarget().getPosition();
     if (!marca[j]){
        dfs(j, grafo, marca);
     }
}
```




```
largo con: 1
1
5
6
8
7
largo con: 2
2
3
4
```

¿Cuántas veces visitamos cada vértice?

¿Cómo lo garantizamos?

Encontrar los vuelos con un costo determinado

Dado un Grafo orientado pesado, como el de la figura, implementar un método que retorne una lista con **todos los caminos** cuyo **costo total** sea igual a **10**. Se considera ***costo total del camino*** a la suma de los costos de las aristas que forman parte del camino, desde un vértice origen a un vértice destino.

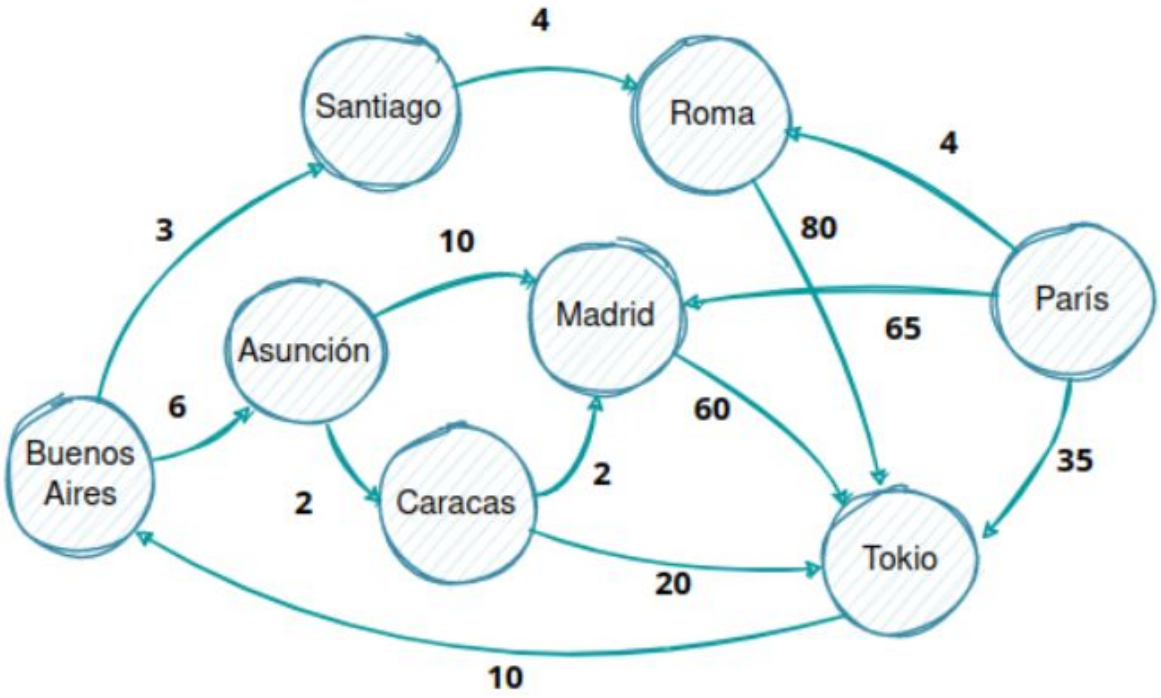
Se recomienda implementar un método público que invoque a un método recursivo privado.

¿Cuáles son resultados posibles?

Bs As - Asunción - Caracas - Madrid

Asunción - Madrid

Tokio - Bs As



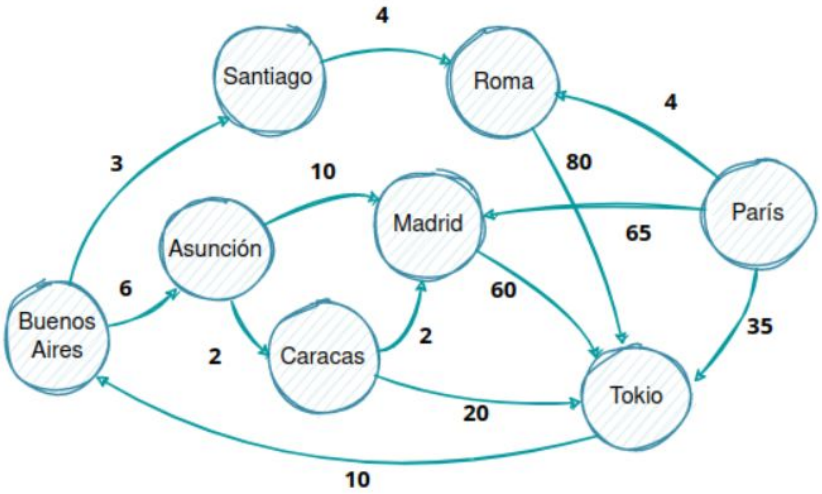
Encontrar los vuelos con un costo determinado

```
public class Vuelos<T> {

 public List<List<String>> dfsConCosto(Graph<T> grafo) {
   boolean[] marca = new boolean[grafo.getSize()];
   List<String> lis = null;
   List<List<String>> recorridos = new LinkedList<LinkedList<String>>();
   int costo = 0;
   for (int i=0; i<grafo.getSize(); i++) {
       lis = new LinkedList<>();
       dfsConCosto(i, grafo, lis, marca, costo, recorridos);
   }
   return recorridos;
  }

 private void dfsConCosto(int i, Graph<T> grafo, List<String> lis,boolean[] marca, int costo,
             List<List<String>> recorridos) {
  //TODO
 }
}
```

¿Cuántas veces visitamos cada vértice?

Encontrar los vuelos con un costo determinado

```
private void dfsConCosto(int i, Graph<T> grafo, List<String> lis,boolean[] marca, int costo,
                         List<List<String>> recorridos) {
   Vertex<T> v = grafo.getVertex(i);
   lis.add(v.getData().toString());
   marca[i] = true;
   if ((costo) == 10)
    // Caso base: se alcanza el costo 10, se guarda el recorrido
       recorridos.add(new LinkedList<String>(lis));
   else{
     List<Edge<T>> ady = grafo.getEdges(v);
     for (Edge<T> e: ady) {
       j= e.getTarget().getPosition();
       if (!marca[j]) {
           p = e.getWeight();
           if ((costo + p) <= 10)
                   dfsConCosto(j, grafo, lis, marca, costo + p, recorridos);
       }
     }
    }
  // Backtracking: desmarcar y eliminar el último vértice del camino
  lis.remove(v.getData().toString());
  marca[i] = false;
}
```



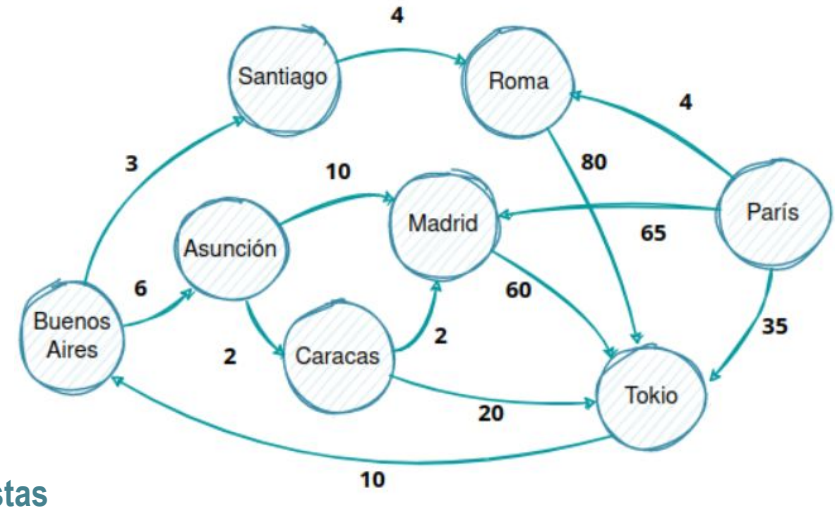
Analicemos los parámetros: ¿Qué mejoras se podrían hacer?

Encontrar los vuelos con un costo determinado

```java
public class VuelosPrueba {
   public static void main(String[] args) {
        Graph<String> grafo = new AdjListGraph<>();
        grafo.createVertex( "Buenos Aires"); // 0
        grafo.createVertex( "Santiago"); // 1
        grafo.createVertex( "Asunción"); // 2
        grafo.createVertex( "Roma"); // 3
        grafo.createVertex( "Madrid"); // 4
        grafo.createVertex( "Caracas"); // 5
        grafo.createVertex( "París"); // 6
        grafo.createVertex( "Tokio"); // 7
        grafo.connect(grafo.getVertex( 0), grafo.getVertex( 1),3);
        grafo.connect(grafo.getVertex( 0), grafo.getVertex( 2),6);
        grafo.connect(grafo.getVertex( 1), grafo.getVertex( 3),4);
        grafo.connect(grafo.getVertex( 2), grafo.getVertex( 4),10);
        grafo.connect(grafo.getVertex( 2), grafo.getVertex( 5),2);
        grafo.connect(grafo.getVertex( 3), grafo.getVertex( 7),80);
        grafo.connect(grafo.getVertex( 4), grafo.getVertex( 7),60);
        grafo.connect(grafo.getVertex( 5), grafo.getVertex( 4),2);
        grafo.connect(grafo.getVertex( 5), grafo.getVertex( 7),20);
        grafo.connect(grafo.getVertex( 6), grafo.getVertex( 3),4);
        grafo.connect(grafo.getVertex( 6), grafo.getVertex( 4),65);
        grafo.connect(grafo.getVertex( 6), grafo.getVertex( 7),35);
        grafo.connect(grafo.getVertex( 7), grafo.getVertex( 0), 10);
        Vuelos<String> vuelos = new Vuelos <>();
        List<List<String>> lista=vuelos.dfsConCosto(grafo);
        System.out.println( "Recorridos con costo 10:" );
        for (List<String> l : lista)
            System.out.println(l);
   }
}
```

Crear Vértices

Crear Aristas




```
Recorridos con costo 10:
[Buenos Aires, Asunción, Caracas, Madrid]
[Asunción, Madrid]
[Tokio, Buenos Aires]
```

Grafos
BFS (Breath First Search)

El algoritmo BFS es la generalización del **recorrido por niveles** de un árbol.
La estrategia es la siguiente:

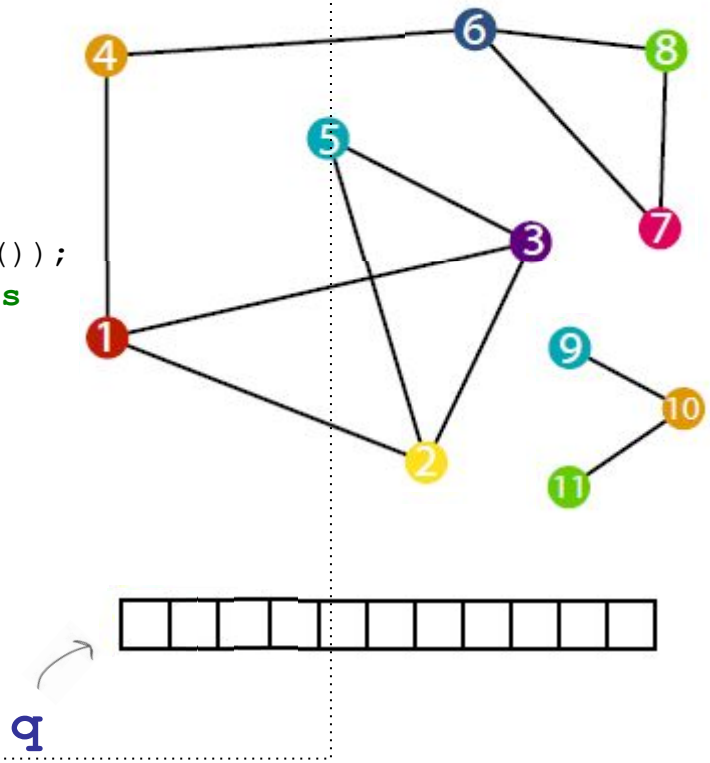
- Partir de algún vértice v, visitar v, después visitar cada uno de los vértices adyacentes a v.
- Repetir el proceso para cada vértice adyacente a v, siguiendo el orden en que fueron visitados.

Si desde v no fueran alcanzables todos los nodos del grafo: elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

```
public class Recorridos {
  public void bfs(Graph<T>grafo) {
      boolean[] marca = new boolean[grafo.getSize()];
      for (int i = 0; i < grafo.getSize(); i++) {
        if (!marca[i]) {
          this.bfs(i, grafo, marca);
        }
      }
  }
  private void bfs(int i, Graph<T> grafo, boolean[] marca) {
  //TODO
  }
}
```

Grafos
BFS (Breath First Search)

```
private void bfs(int i, Graph<T> grafo, boolean[] marca) {

    Queue<Vertex<T>> q = new Queue<Vertex<T>>();
    q.enqueue(grafo.getVertex(i));
    marca[i] = true;

    while (!q.isEmpty()) {
        Vertex<T> w = q.dequeue();
        System.out.println("Visito vértice: "+ w.getData());
        // visitar todos los adyacentes de w no visitados
        List<Edge<T>> adyacentes = grafo.getEdges(w);
        for (Edge<T> e: adyacentes) {
            int j = e.getTarget().getPosition();
            if (!marca[j]) {
                marca[j] = true;
                q.enqueue(e.getTarget());
            }
        }
    }
}
```



Ejercicio de Parcial

Tiempo de infección de una red

Un poderoso e inteligente **virus de computadora infecta** cualquier computadora en **1 minuto**, logrando infectar toda la red de una empresa de cientos de computadoras.

Dado un **grafo** que representa la **red o conexiones entre las computadoras** de la empresa, y una **computadora infectada**, escribir un programa en Java que **determine el tiempo** que demora el virus en **infectar** el resto de las computadoras.

Todas las computadoras pueden ser infectadas, no todas las computadoras tienen conexión directa entre sí, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.

¿Qué significa que "no todas las computadoras tienen conexión directa entre sí"?

¿Qué significa que "un mismo virus puede infectar un grupo de computadoras al mismo tiempo"?

Ejercicio de Parcial

Tiempo de infección de una red

```
public static int calcularTiempoInfeccion(Graph<String> grafo, Vertex<String> inicial)
{
    int tiempo = 0;
    boolean visitados[] = new boolean[grafo.getSize()];
    Queue<Vertex<String>> cola = new Queue<Vertex<String>>();
    visitados[inicial.getPosition()] = true;
    cola.enqueue(inicial);
    cola.enqueue(null);
    while (!cola.isEmpty()) {
        Vertex<String> v = cola.dequeue();
        if (v != null) {
            List<Edge<String>> adyacentes = grafo.getEdges(v);
            for (Edge<String> e : adyacentes) {
                Vertex<String> w = e.getTarget();
                if (!visitados[w.getPosition()]) {
                    visitados[w.getPosition()] = true;
                    cola.enqueue();
                }
            }
        } else if (!cola.isEmpty()) {
            tiempo++;
            cola.enqueue(null);
        }
    }
    return tiempo;
}
```