2️⃣

Modulo 2

Temario

1. Terminologia y representaciones de grafos
2. Grafos en java
3. Recorridos
4. Sort topologico
5. Caminos de costo minimo
6. Arbol de expansion minimo

Notas

- Buena practica `ciudades != null && !ciudades.esVacio()`
- Prestar atencion a cuando utilizar `ListaGenerica<Vertice<T>>` y cuando `ListaGenerica<T>` en recorridos
- Cuando es necesario realizar un recorrido desde un nodo en puntual, recordar que se debe buscar el nodo entre la lista de vertices

```
// metodo disparador del dfs

vertices.comenzar();
Vertice<String> vInicial = null;

while (!vertices.fin() && vInicial == null) { // compruebo que no se encontro
  Vertice<String> vertice = vertices.proximo(); // obtengo vertice
  if (vertice.dato().equals(origen)) { // equals SI O SI, no se puede usar "=="
    vInicial = vertice; // vertice desde donde iniciamos el recorrido
  }
}
if (vInicial != null) {
  // llamado al dfs
}
```

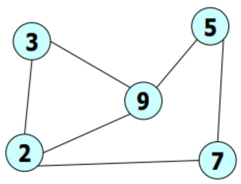
Grafos

- Digrafo → aristas como par ordenado `(u, v)`.
  Grafo no dirigido → aristas como par **no** ordenado `{u, v}`.
- Existen grado de salida *grado_out* y grado de entrada *grado_in*.
- Camino simple en dirigidos = camino en donde no se repiten vertices.
- Longitud de un ciclo = cantidad de aristas involucradas.
- Bucle = ciclo de longitud 1

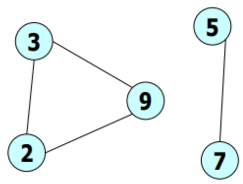
- Grafo ponderado/pesado/con costo.

Conectividad

- No dirigidos: bosque (aciclico) → arbol libre (conexo) → arbol (nodo raiz)

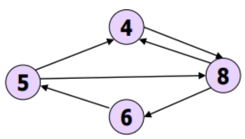


Conexo

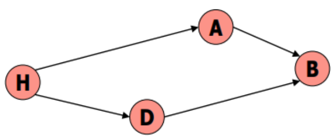


No Conexo

- Dirigidos: todos los vertices tienen `[grado_out, grado_in] > 1`



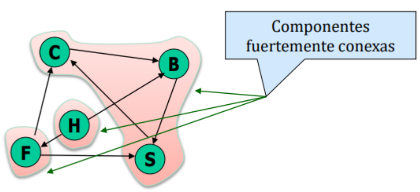
Fuertemente Conexo



No Fuertemente Conexo
Débilmente Conexo

> 💡 Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es débilmente conexo.

- Componente conexa = Subgrafo conexo maximal (mayor conexion posible)



No Fuertemente Conexo

Propiedades

✓*Siempre:* $m \leq (n*(n-1))/2$

✓*Si G conexo:* $m \geq n-1$
✓*Si G árbol:* $m = n-1$
✓*Si G bosque:* $m \leq n-1$

n vertices y m arcos

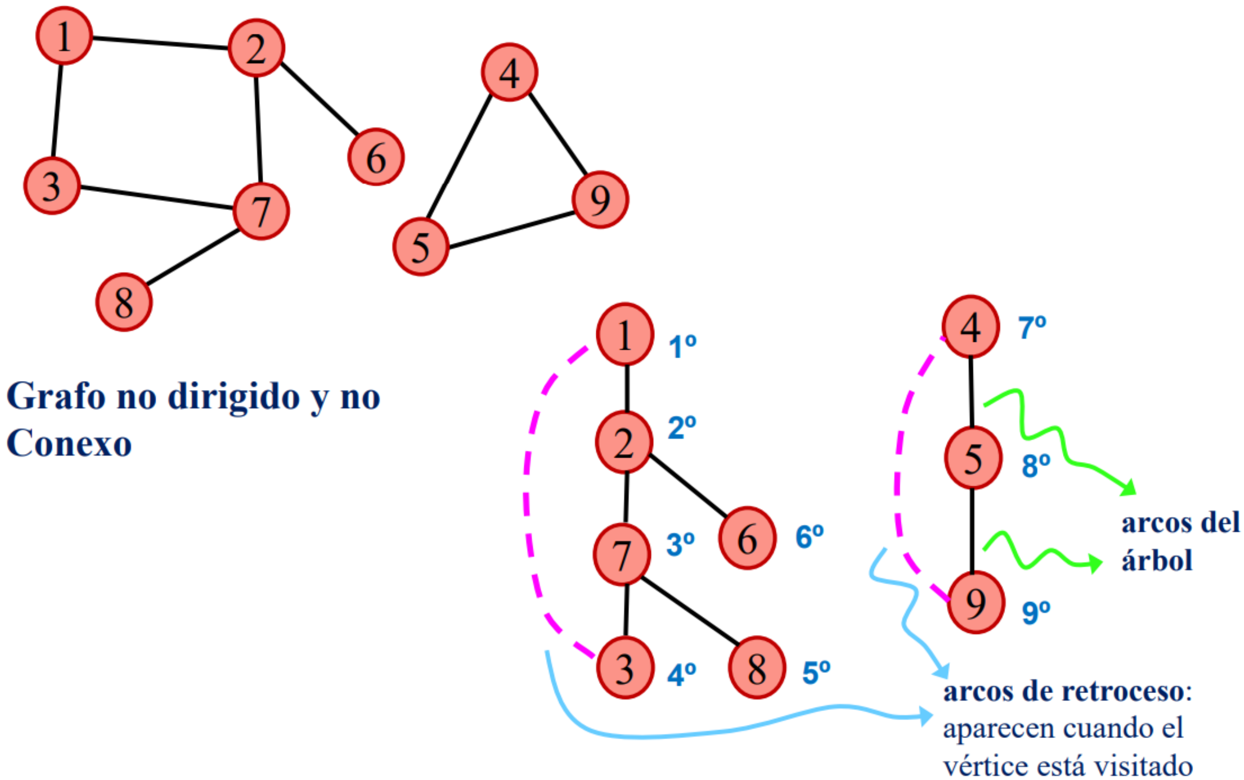
Representaciones

- Matriz de adyacencia
  memoria = `O(V²)`
  acceso = `O(1)`
  util para grafos densos `E` se aproxima a `V²`
- Lista de adyacencia
  memoria = `O(V + E)`
  acceso = `O(grado del grafo)` ≤ `O(V)`

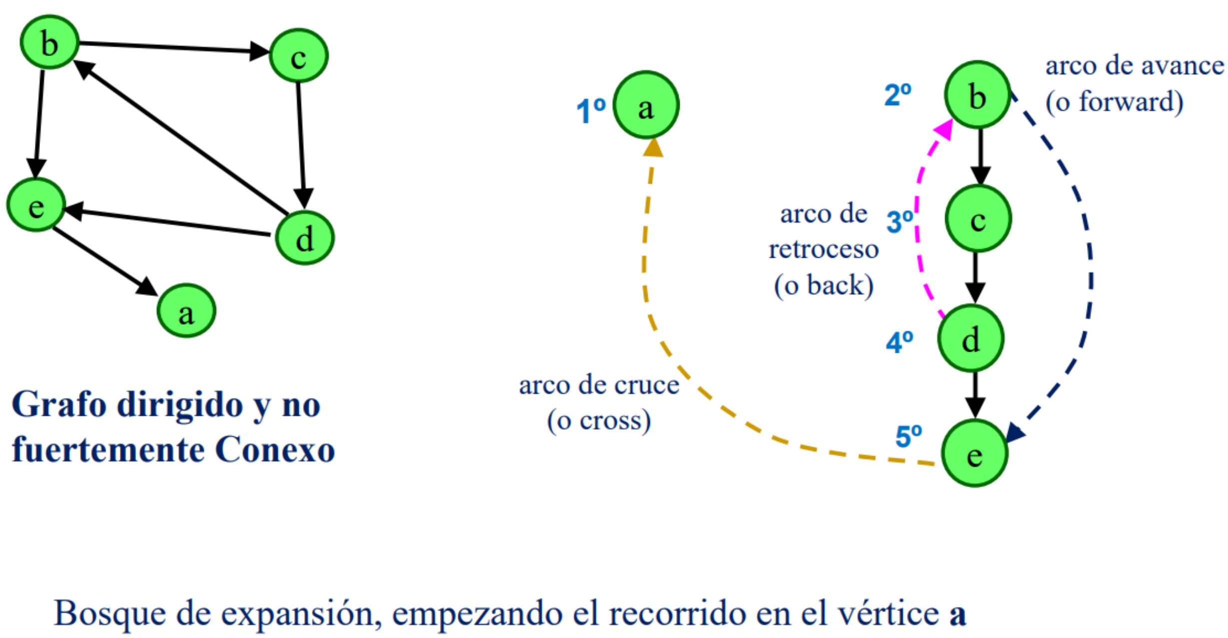
Recorridos

- DFS
  - generalizacion del preorden en un arbol.
  - No es unico, depende del orden de aparicion de los nodos en la representacion.
  - Se considera cada dfs y el bucle en main `O(V + E)`
  - Arbol de expansion/abarcador
  - Aplicaciones: Mismo orden que los algoritmos de recorrido.
    - Encontrar las componentes **conexas** de un grafo **no dirigido**.
    - Prueba de aciclicidad (arbol de expansion sin arcos).
    - Encontrar las componentes **fuertemente conexas** de un grafo **dirigido**.
- BFS
  - generalizacion del recorrido por niveles

Arbol de expansion



Ejemplo 1



Ejemplo 2

Algoritmo de Kosaraju `O (V + E)`

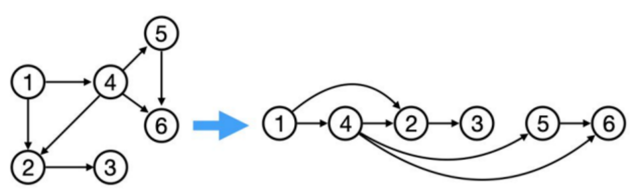
Pasos:

1. Aplicar DFS(G) rotulando los vértices de G en post-orden (apilar).
2. Construir el grafo reverso de G, es decir G^R (invertir los arcos).
3. Aplicar DFS (G^R) comenzando por los vértices de mayor rótulo (tope de la pila).
4. Cada árbol de expansión resultante del paso 3 es una componente fuertemente conexa.

Si resulta un único árbol entonces el digrafo es fuertemente conexo.

Orden topologico

- Grafos Dirigidos Aciclicos (DAG)
- Orden topologico **no** es unico.
- Ordenación horizontal de los vértices, con los arcos de izquierda a derecha.



- Aplicaciones:
  - Indicar presedencia de eventos.
  - Planificacion de tareas.
  - Organizacion curricular.
- Versiones:

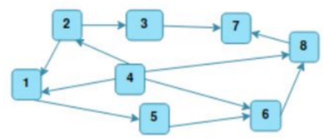
  1️⃣ `O(V² + E)` Arreglo en el que se almacenan los **grados de entradas** de los vértices y en cada paso se toma de allí un vértice con grado_in = 0, se "**eliminan**" sus conexiones, y continua.

  2️⃣ `O(V + E)` Se utiliza una pila/cola para almacenar vertices con grado_in = 0

  3️⃣ Aplicando DFS
  - Numerando los vertices
  - Apilando los vertices

```java
public class SortTopologico<T> {

 public PilaGenerica<Vertice<T>> sortTopologico(Grafo<T> grafo) {
      boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()];
      PilaGenerica<Vertice<T>> pila = new PilaGenerica<Vertice<T>>();
      for (int i = 0; i < grafo.listaDeVertices().tamanio(); i++) {
           if (!marca[i])
                this.sortTopologico(i, grafo, pila, marca);
      }
      return pila;
 }
 private void sortTopologico(int i,Grafo<T> grafo,PilaGenerica<Vertice<T>> pila,boolean[] marca){
      marca[i] = true;
      Vertice<T> v = grafo.listaDeVertices().elemento(i);
      ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
      ady.comenzar();
      while (!ady.fin()) {
           Arista<T> a = ady.proximo();
           if (!marca[a.getVerticeDestino().getPosicion()]) {
                int j = a.getVerticeDestino().getPosicion();
                this.sortTopologico(j, grafo, pila, marca);
           }
      }
      pila.apilar(v);
    }
}
```




```
Console × Problems
<terminated> GrafoTest (1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (2
El Orden Topológico encontrado es: 4 -> 2 -> 3 -> 1 -> 5 -> 6 -> 8 -> 7
```

Codigo de implementacion

Algoritmos de caminos minimos

- Longitud del camino **no** pesado = N° de aristas
- Casos y estrategias:
  - Grafos sin peso
    - Recorrido en amplitud basado en BFS
  - Grafos con peso > 0
    - Dijkstra `O(V² + E)`
    - Dijkstra + Heap `O(E*logV)`
    - Dijkstra + Heap + Insercion del vertice modificado `O(E*logV)`
  - Grafos con peso
  - Grafos dirigidos aciclicos `O(V + E)`
- Floyd Algorithm `O(V³)` para caminos minimos entre todos los pares de vertices.
  - Lleva matriz de costos minimos y matriz de vertices intermedios

Grafos	BFS O(V+E)	Dijkstra O(E log V)	Algoritmo modificado (encola vértices) O(V*E)	Optimización de Dijkstra (sort top) O(V+E)
No pesados	Óptimo	Correcto	Malo	Incorrecto si tiene ciclos
Pesados	Incorrecto	Óptimo	Malo	Incorrecto si tiene ciclos
Pesos negativos	Incorrecto	Incorrecto	Óptimo	Incorrecto si tiene ciclos
Grafos pesados acíclicos	Incorrecto	Correcto	Malo	Óptimo

Correcto: adecuado pero no el mejor
Malo: una solucion muy lenta

▼ Pseudocodigos

```
Camino_min_GrafoNoPesadoG,s) {
    para cada vértice v Є V
        D_v = ∞; P_v = 0;
    D_s = 0; Encolar (Q,s);
    Mientras (not esVacio(Q)) {
        Desencolar(Q,u);
        para c/vértice w Є V adyacente a u {
            si (D_w = ∞) {
                    D_w = D_u + 1;
                    P_w = u;
                    Encolar(Q,w);
            }
        }
    }
}
```

```
Dijkstra(G,w, s){
    para cada vértice v Є V
        D_v = ∞;    P_v = 0;
    D_s = 0;
    para cada vértice v Є V {
        u = vérticeDesconocidoMenorDist;
        Marcar u como conocido;
        para cada vértice w Є V adyacente a u
            si (w no está conocido)
                si (D_w > D_u + c(u,w)) {
                    D_w = D_u + c(u,w);
                    P_w = u;
                }
        }
    }
}
```

```
Camino_min_GrafoPesosPositivosyNegativosG,s) {
    Ds = 0; Encolar (Q,s);
    Mientras (not esVacio(Q)) {
       Desencolar(Q,u);
       para c/vértice w ∈ V adyacente a u {
           si (Dw > Du + c(u,w)) {
                  Dw = Du + c(u,w);
                  Pw = u;
                  si (w no está en Q)
                      Encolar(Q,w);
)          }
)       }
)   }
 }
```

```
Camino_min_GrafoDirigidoAcíclico(G,s){

      Ordenar topológicamente los vértices de G;

      Inicializar Tabla de Distancias(G, s);

      para c/vértice u del orden topológico
         para c/vértice w ∈ V adyacente a u
              si (Dw > Du + c(u,w)) {
                    Dw = Du + c(u,w);
                    Pw = u;
              }
 }
```

Toma cada vértice como intermedio, para calcular los caminos

```
para k=1 hasta cant_Vértices(G)
     para i=1 hasta cant_Vértices(G)
       para j=1 hasta cant_Vértices(G)
          si (D[i,j] > D[i,k] + D[k,j]) {
              D[i,j] = D[i,k] + D[k,j];
              P[i,j] = k;
          }
```

Distancia entre los vértices *i* y *j*, pasando por *k*

Arbol de expansion minima

El árbol de expansión mínima es un árbol formado por las aristas de G que conectan todos los vértices con un costo total mínimo.