Sesión 4 (6ª Semana)

- Isaac Lozano (isaac.lozano@urjc.es)
- Raúl Martín (raul.martin@urjc.es)
- Sergio Salazar (s.salazarc.2018@alumnos.urjc.es)
- Francisco Tórtola (f.tortola.2018@alumnos.urjc.es)
- Cristian Pérez (c.perezc.2018@alumnos.urjc.es)
- Xuqiang Liu (x.liu1.2020@alumnos.urjc.es)
- Alicia Pina (a.pinaz.2020@alumnos.urjc.es)
- Sara García (s.garciarod.2020@alumnos.urjc.es)
- **Raúl Fauste** (r.fauste.2020@alumnos.urjc.es)

Resultados AdaByron 2022

Pos.	Equipo	Puntuación		A	B	C	D	E	F	G	H	I	J	K	L	M
7	Teamto de Verano	7	786	1/87			7/228	4/36		1/28	1/51	3/32	3/64		12	
9	Cean	5	433	1/88				1/31		4/69	5	1/105	2/60			4
16	LongLongLovers	4	558					1/54		5/208		1/108	2/88		2	3
17	ReyEmeriteam	4	620					1/50		1/248		2/140	3/122			
21	πk2's	4	758	1			1/-1	5/113		2/100		3/220	3/145			3
25	C-ANSInos	3	259	1				1/53		2	3	1/175	1/11			
31	DTX	3	385	4				1/118		12		2/110	1/97			

45 equipos – 3 equipos en fase nacional

1º Categoría de primero (9) 2º Categoría de segundo (16) 3º de Categoría de segundo (17)

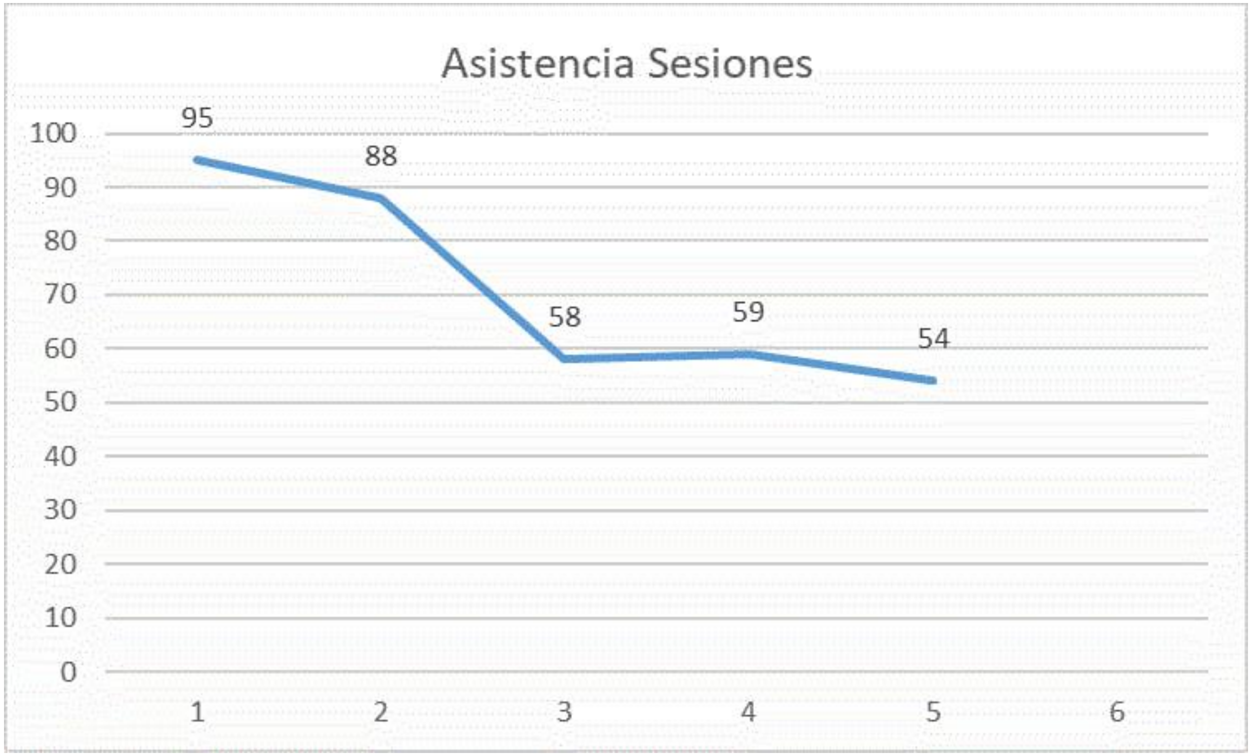
Resultados AdaByron 2023

RANK		TEAM	SCORE		A	B	C	D	E	F	G	H	I	J	K
4		πk2s Universidad Rey Juan Carlos	7	1131	286 2 tries	195 1 try	277 2 tries					26 1 try	19 1 try	103 3 tries	125 2 tries
7		Teamto de Verano Universidad Rey Juan Carlos	5	549	1 try	226 2 tries						15 1 try	69 1 try	34 2 tries	145 2 tries
20		MIIDAS AC? Universidad Rey Juan Carlos	3	321	1 try	1 try			2 tries			70 1 try	131 4 tries	60 1 try	3 tries
22		DPrimidos Universidad Rey Juan Carlos	3	481	1 try							56 1 try	151 4 tries	194 2 tries	3 tries
26		while(true) {siesta()} Universidad Rey Juan Carlos	2	215								41 1 try		154 2 tries	7 tries
27		LongLongLovers Universidad Rey Juan Carlos	2	228						2 tries		41 2 tries	1 try	127 3 tries	8 tries
38		PCVita Universidad Rey Juan Carlos	1	91	3 tries							91 1 try	10 tries		

44 equipos - ¿1 equipo al nacional?
4º Categoría general

Asistencia AdaByron 2023

54 personas con asistencia registrada

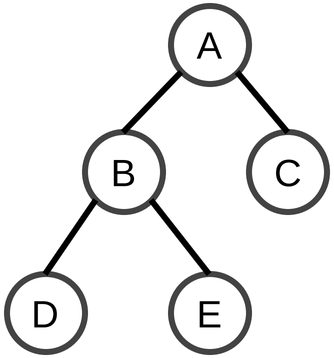


Contenidos

- Grafos
  - Definición y Representación
  - Recorrido Anchura y Profundidad (BFS, DFS)
  - Componentes Conexas
  - Ordenamiento Topológico
  - Puntos de articulación
  - Bipartitos

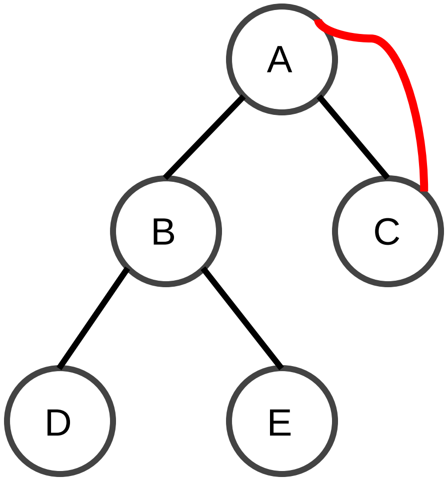
Grafos

- Árboles: representan relaciones jerárquicas
  - Tienen un padre (excepto la raíz)
  - Pueden tener hijos



Grafos

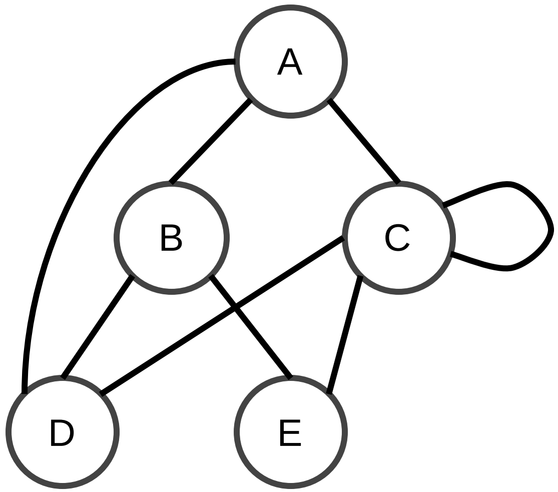
- Restricciones jerárquicas:
  - No admite ciclos



- C no puede ser padre de su padre (A)

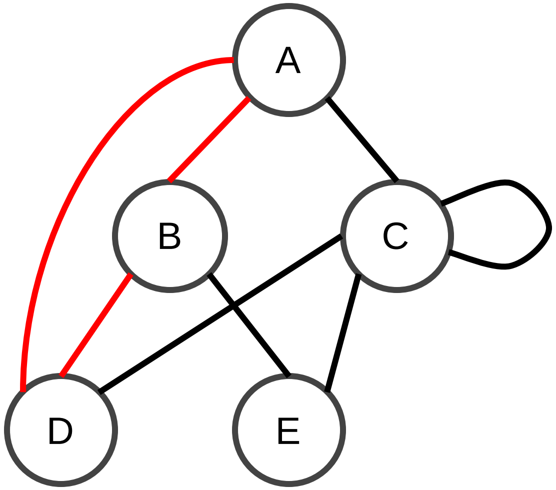
Grafos

- Grafos: mayor libertad para representar un sistemas y sus relaciones/interacciones



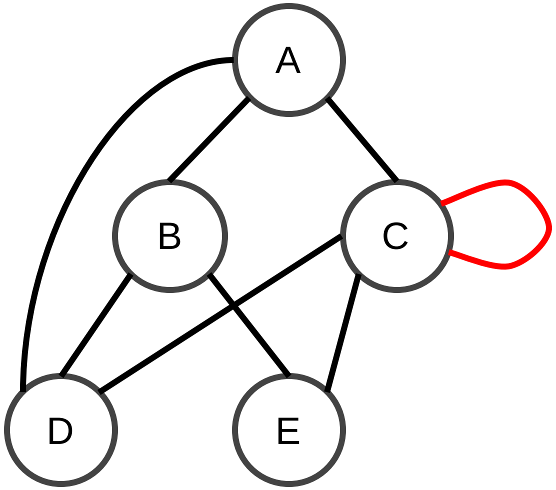
Grafos

- podemos tener ciclos



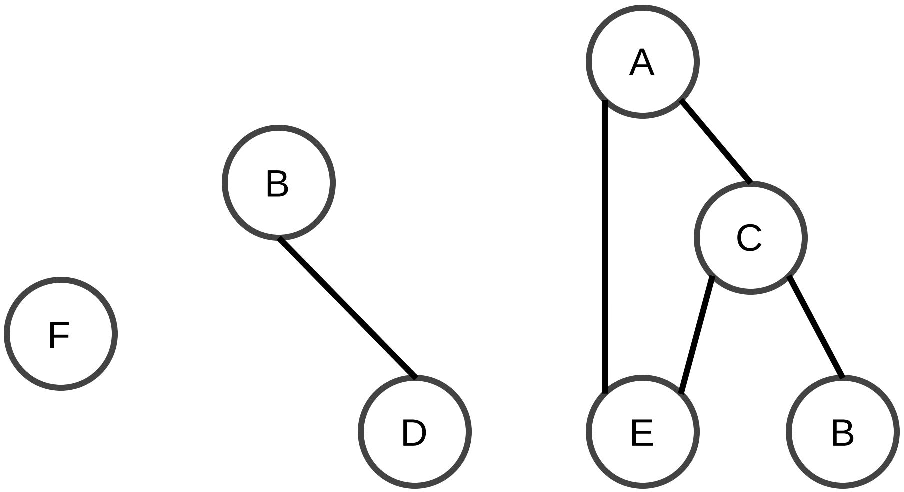
Grafos

- podemos tener bucles sobre el mismo elemento



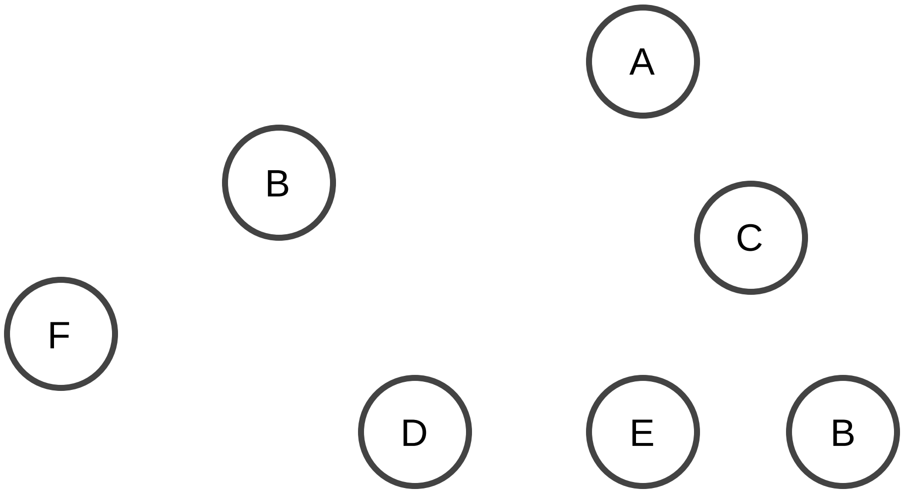
Grafos

- podemos tener grupos aislados en un mismo grafo



Grafos

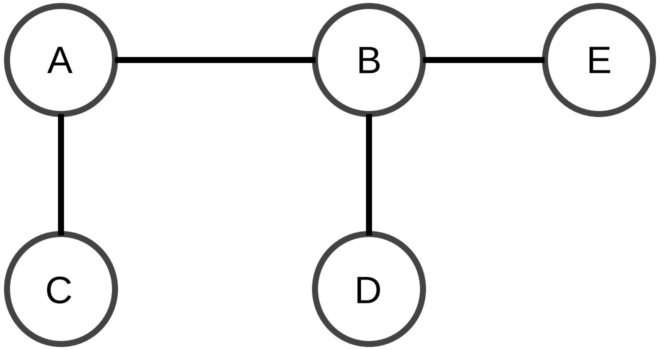
- o elementos totalmente aislados entre sí



Grafos - Definición

- Definición: G = (V,E)
  - Conjunto de vértices: V={A,B,C,D,E}
  - Conjunto de aristas:

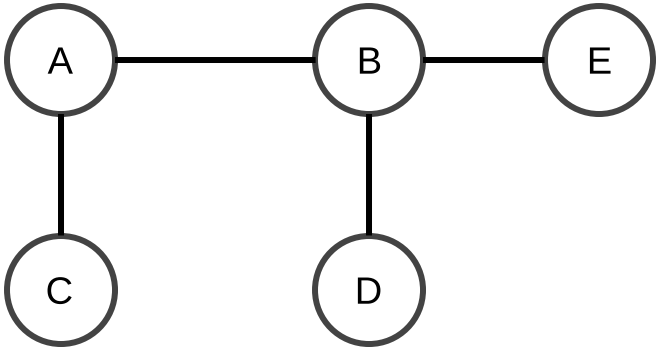
    E={(A,B),(A,C),(B,D),(B,E)}



Grafos – No dirigidos

- Grafo NO dirigido → las aristas **NO** tienen **dirección**

  (A,B) ⇔ (B,A)



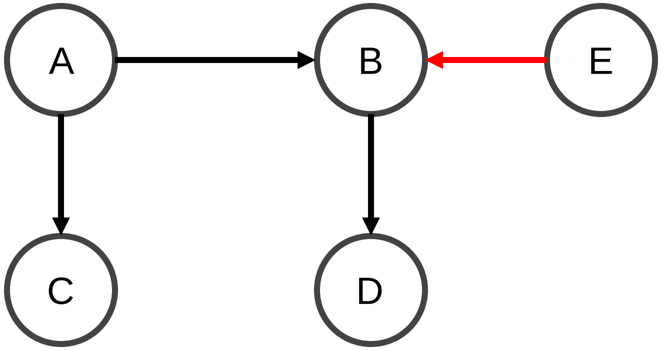
E={(A,B),(B,A),(A,C),(C,A),(B,D),(D,B),(B, E),(E,B)}

Grafos - Dirigidos

- Grafo dirigido → aristas **SI** tienen **dirección**

E={(A,B),(A,C),(B,D),(E, B)}

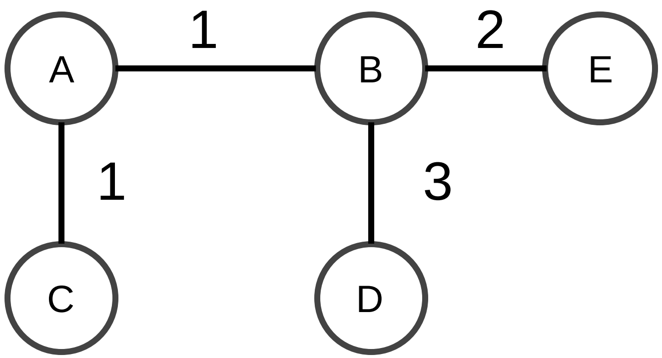
(E,B) ≠ (B,E)



Grafos - Ponderados

- Grafo ponderado → las aristas tienen **pesos/valores**.

E={(A,B,1),(A,C,1),(B,D,3),(B, E,2)}

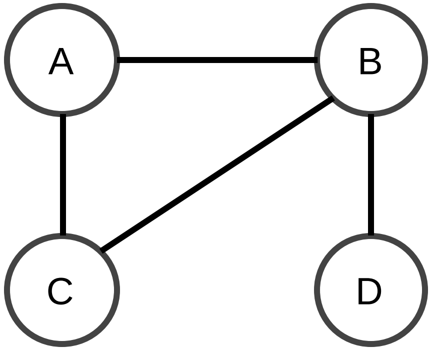


Grafos - Representación

- Representaciones
  - Matriz de adyacencia
  - Lista de adyacencia
  - Lista de aristas

Grafos - Matriz

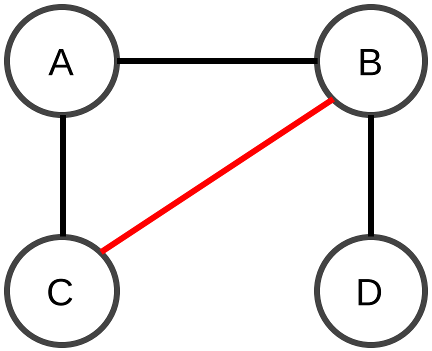
- Matriz de adyacencia
  - Array de dos dimensiones



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Grafos - Matriz

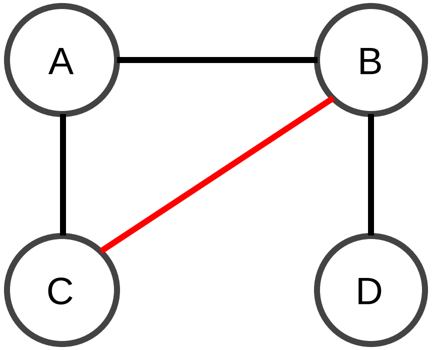
- Arista (B,C) ⇒ m[1][2] = 1



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Grafos - Matriz

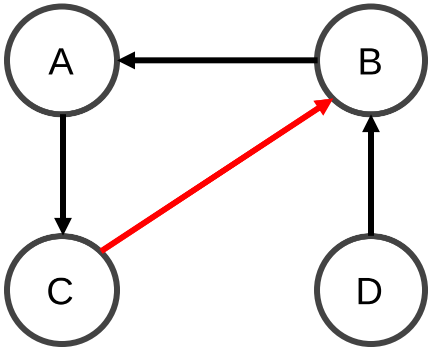
- Matriz **simétrica en grafos no dirigidos**
- m[fila][col]==m[col][fila]



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Grafos - Matriz

- Con **grafo dirigido** la matriz **no** es **simétrica**
- m[2][1]=1 y =m[1][2]=0



	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	1	0	0
D	0	1	0	0

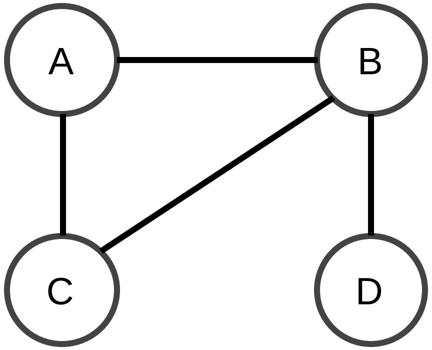
Grafos - Matriz

- Matriz de adyacencia
  - Memoria: $O(|V|^2)$
  - Acceso: O(1)
  - Aristas de un vértice: O(|V|)
    - hay que recorrer toda la fila (incluso si solo tiene una o ninguna)

- Caso de uso: grafos densos (N~=5000)

Grafos – Lista Adyacencia

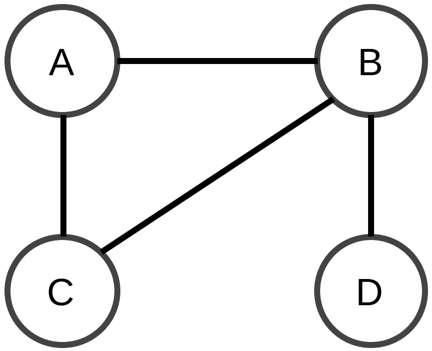
- Lista de adyacencia
  - enumerar las aristas por vértice



A	⇒	{B,C}
B	⇒	{A,C,D}
C	⇒	{A,B}
D	⇒	{B}

Grafos – Lista Adyacencia

- Lista de adyacencia
  - guardar cada lista en un array



A	{B,C}
B	{A,C,D}
C	{A,B}
D	{B}

Grafos – Lista Adyacencia

- Lista de adyacencia
  - Memoria: O(|V|+|E|)
  - Acceso: O(|V|)
    - recorrer todas las aristas de la lista
  - Aristas de un vértice: O(1)
    - en el peor caso tiene aristas a todos los vértices
- útil en grafos dispersos

Grafos – Lista Adyacencia

- Implementación Lista Adyacencia:

```
n = nodos
m = aristas
grafo = vector<int>[n]
//inicializar los vectores
for(i=0; i<m;i++)
        grafo[a].añadir(b)
```

Grafos – Lista Adyacencia

- Implementación Lista Adyacencia:

n = nodos **NO DIRIGIDO**

m = aristas

grafo = vector<int>[n]

//inicializar los vectores

for(i=0; i<m;i++)

grafo[a].añadir(b)

grafo[b].añadir(a)

Grafos - Mapas

- Permite utilizar cualquier tipo de etiquetas
  - no solo índices sino strings u otros

```
traduccion = mapa<string, int>
grafo = vector<int> adyacentes
adyacentes = grafo[traduccion[“madrid”]]
existeArista = adyacentes.contiene(“murcia”)
```

Grafos - Lista Aristas

- Existe otra implementación para representar un grafo
- Más utilizada **para algoritmos específicos**
- Guarda **una lista las conexiones de A a B**
- vector<arista> aristas
- aristas.insertar(arista(a, b))

Grafos - Recorridos

- Recorrer un grafo
  - Recorrido en anchura:

    BFS - Breadth First Search
  - Recorrido en profundidad

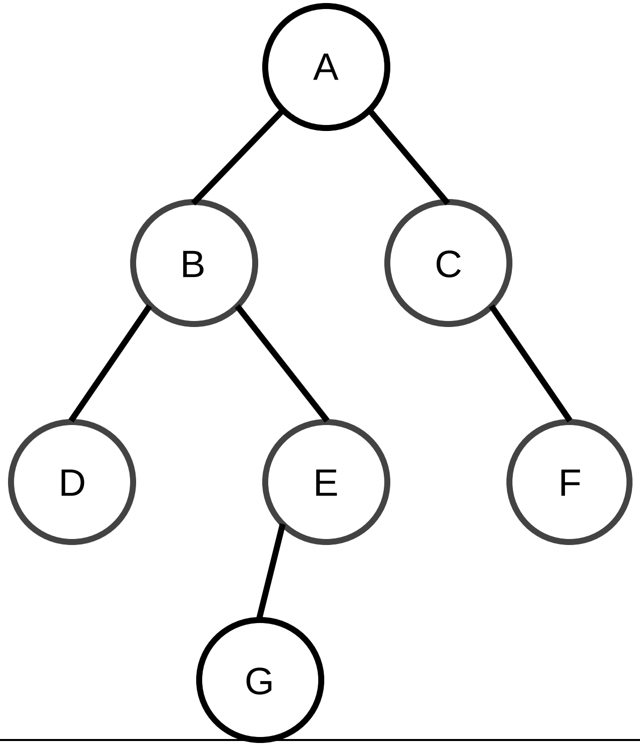
    DFS - Depth First Search

Grafos - Recorridos

- Cosas a tener en cuenta al recorrer:
  - Tenemos que llevar cuenta de por que nodos ya hemos pasado → TLE
  - Cada arista recorrida cuenta como 1 paso
  - El grafo puede tener algún nodo suelto (también hay que recorrerlo)

Grafos - BFS

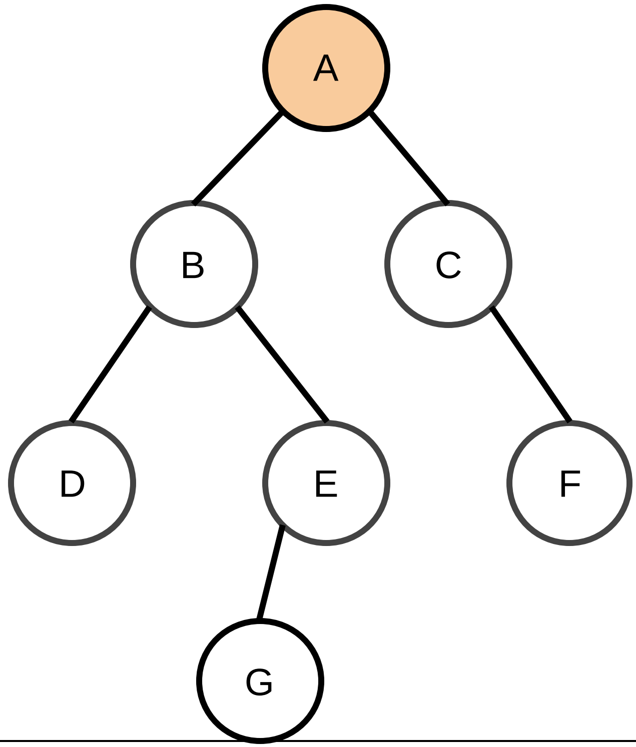
- Recorrido en anchura



Grafos - BFS

- Si empezamos desde A (nodo inicial)
- Recorrido por niveles

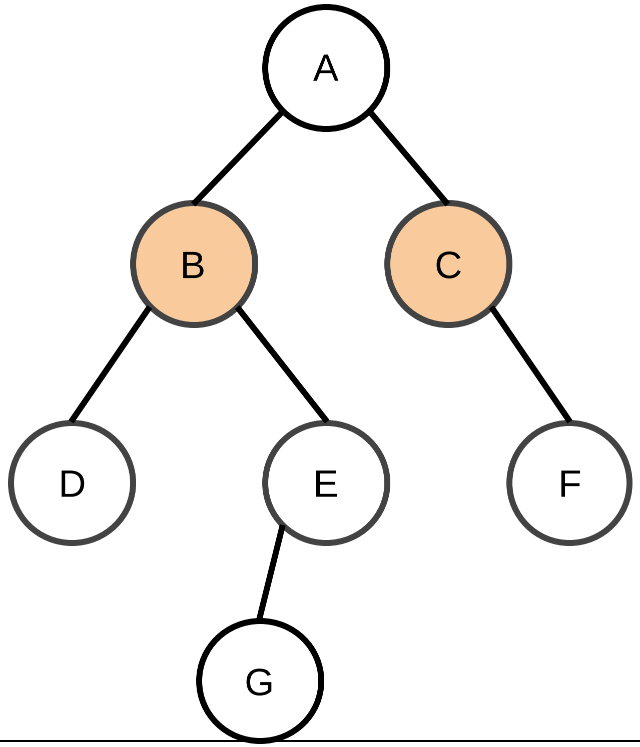
A



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

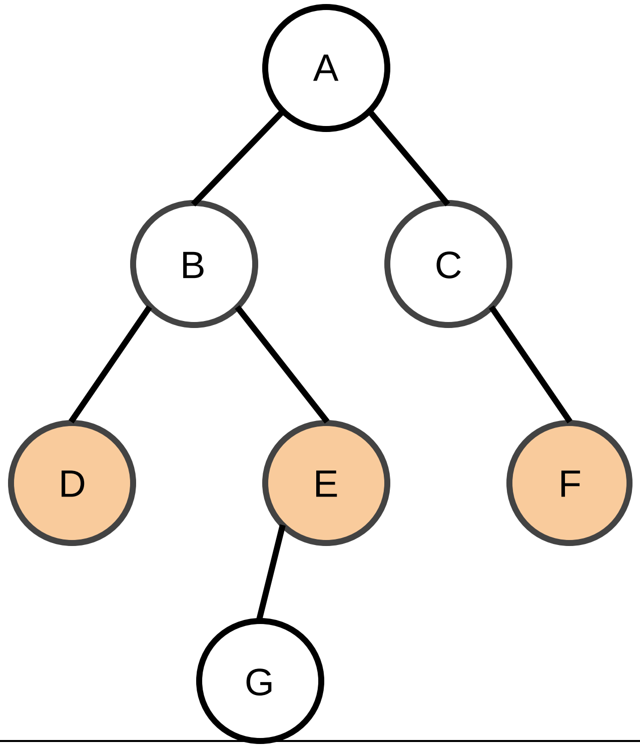
A ⇒ B ⇒ C



Grafos - BFS

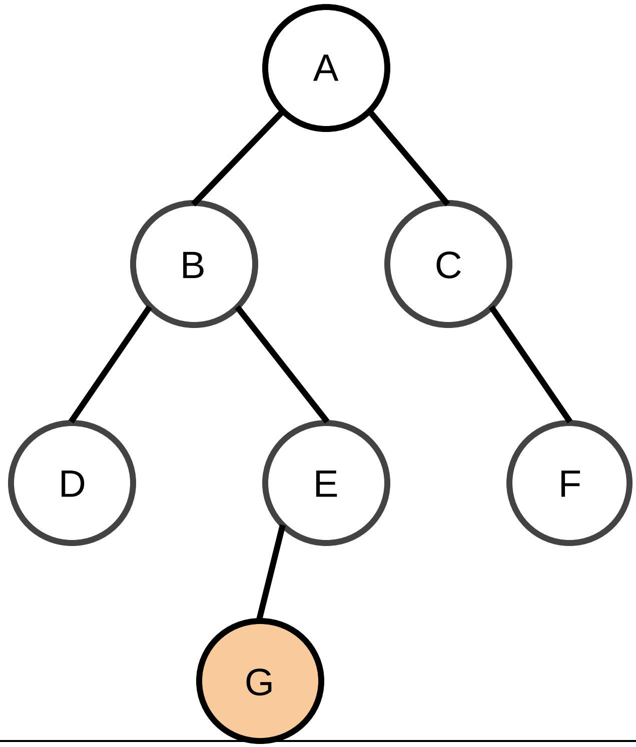
- Si empezamos desde A
- Recorrido por niveles

A ⇒ B ⇒ C ⇒ D ⇒ E ⇒ F



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles



A ⇒ B ⇒ C ⇒ D ⇒ E ⇒ F ⇒ G

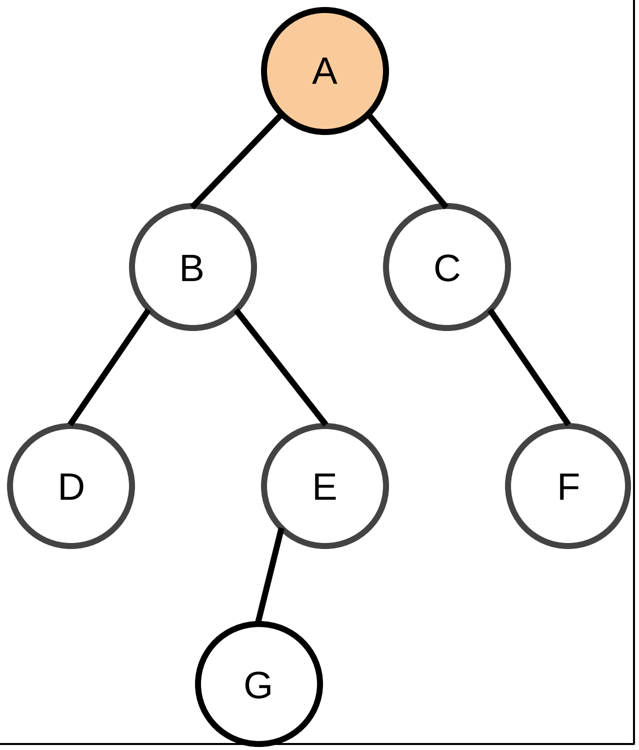
Grafos - BFS

- Implementación
  - Array de valores booleanos (**visitados**)
    - Imprescindible para evitar TLE.
  - **Cola** de vértices a explorar
    - Se procesan en orden de llegada

Grafos - BFS

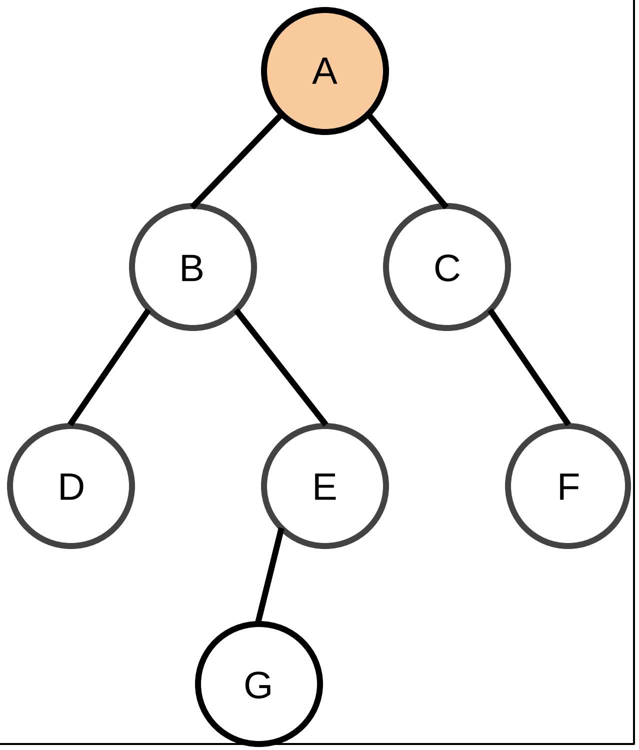
- Inicialización: Elegimos un vértice inicial

```
inicial = 0
visitado[inicial]=true
cola.add(inicial)
```



Grafos - BFS

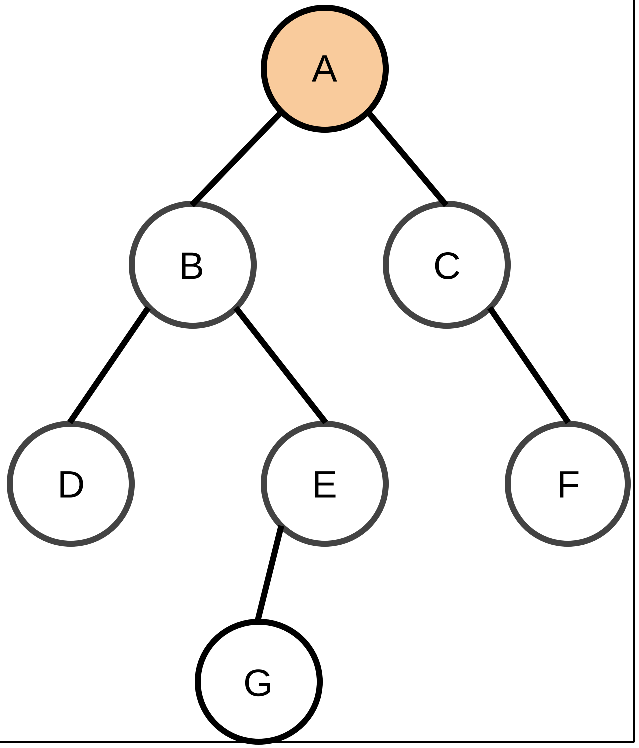
- Cola = {A}
- Visitados = {A}



Grafos - BFS

Recorremos los vértices

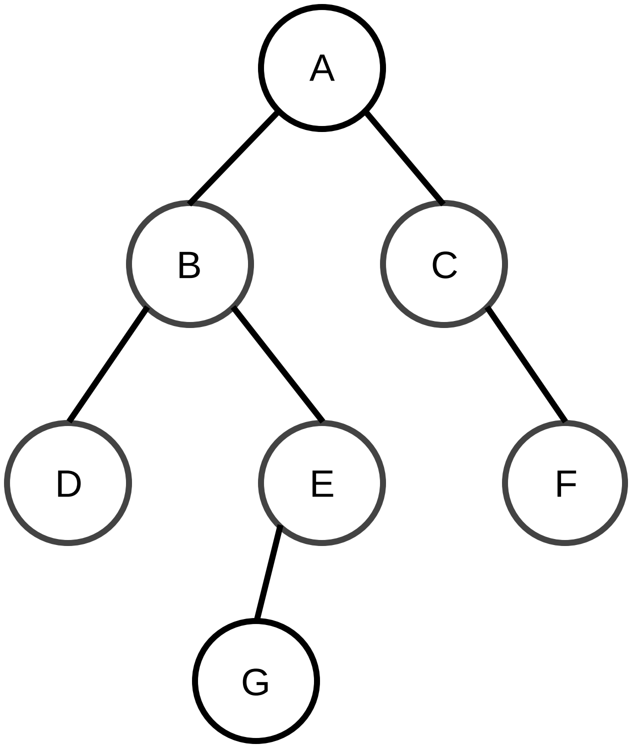
```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

- Cola = {}
- Visitados = {A}
- Sacamos A

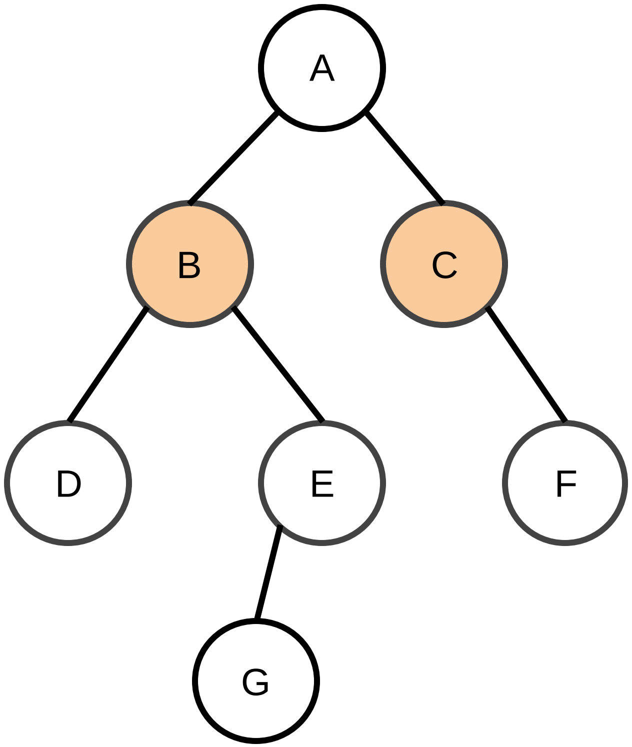
```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

- Cola = {}
- Visitados = {A}

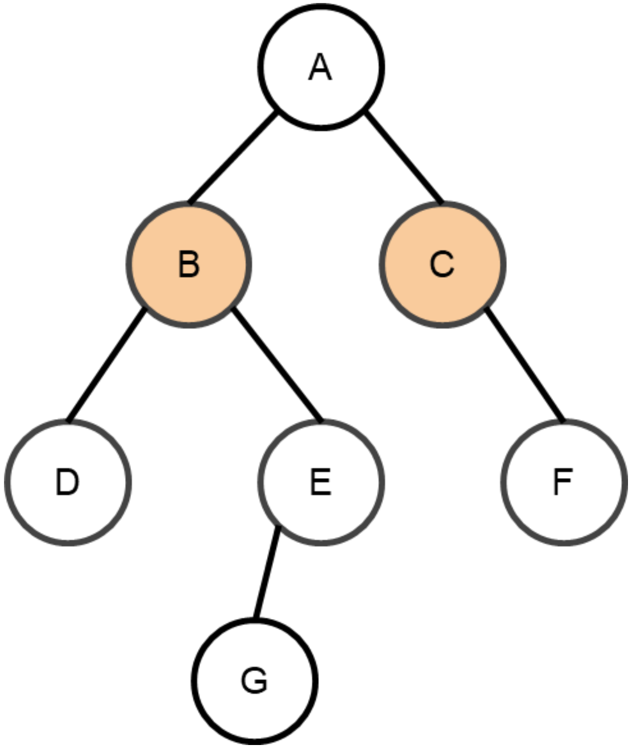
```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

- Cola = {B,C}
- Visitados = {A,B,C}

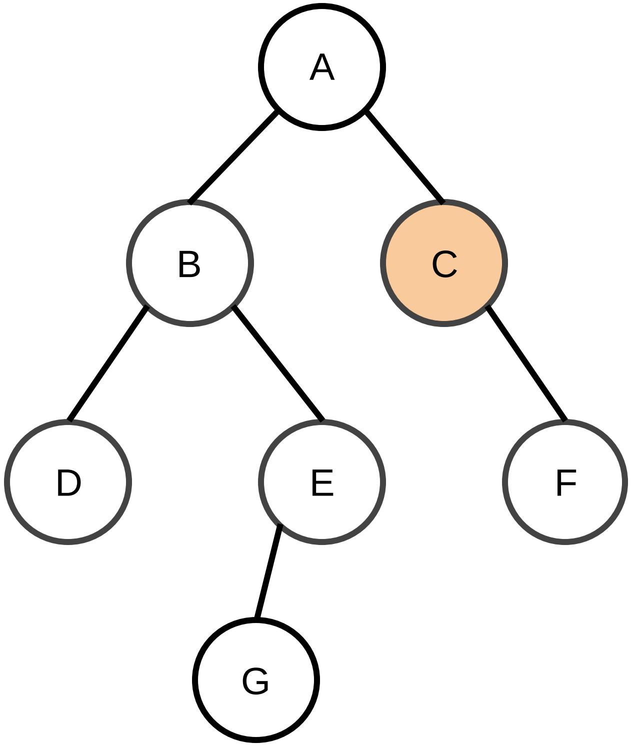
```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

- Cola = {C}
- Visitados = {A,B,C}
- Sacamos B

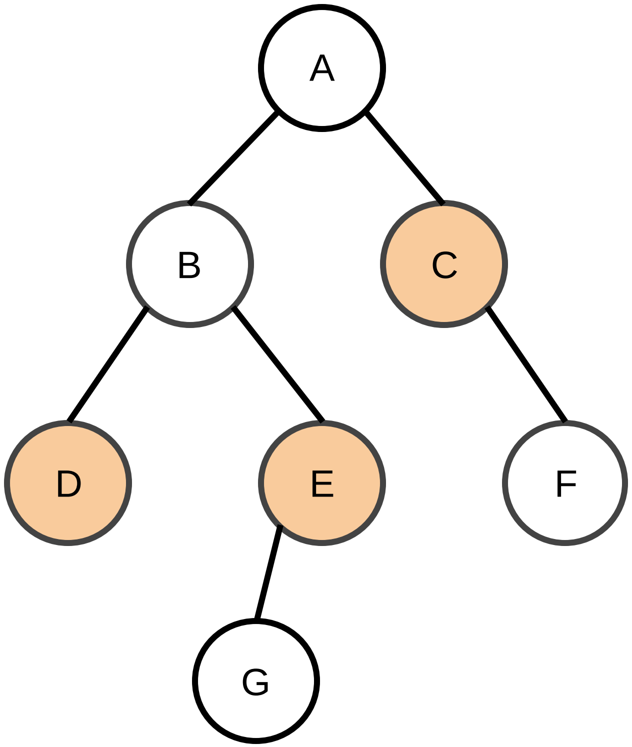
```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

- Cola = {C, D, E}
- Visitados = {A,B,C,D,E}

```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```

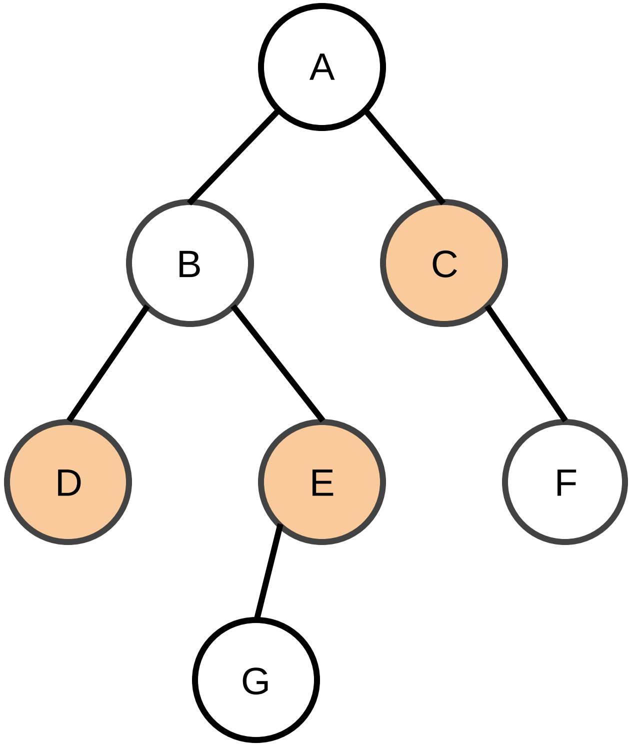


Grafos - BFS

- Cola = {D, E}
- Visitados = {A,B,C,D,E}

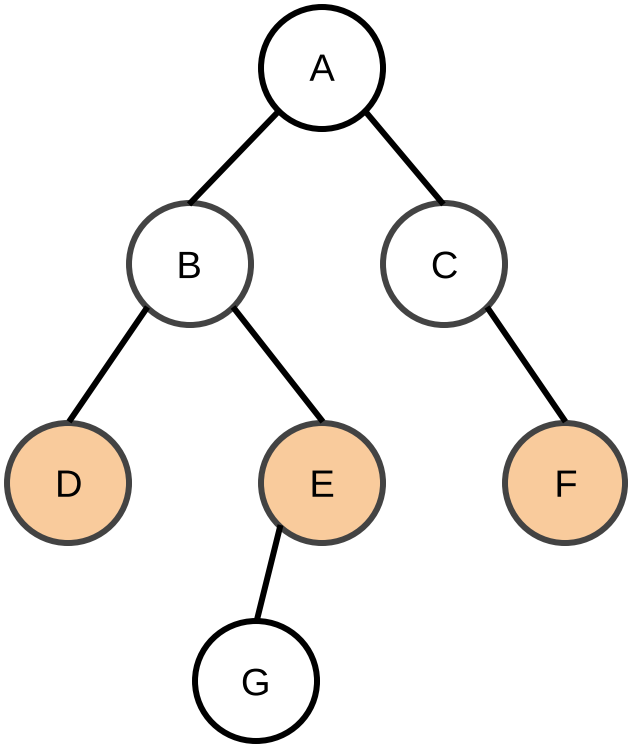
Sacamos C

```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```



Grafos - BFS

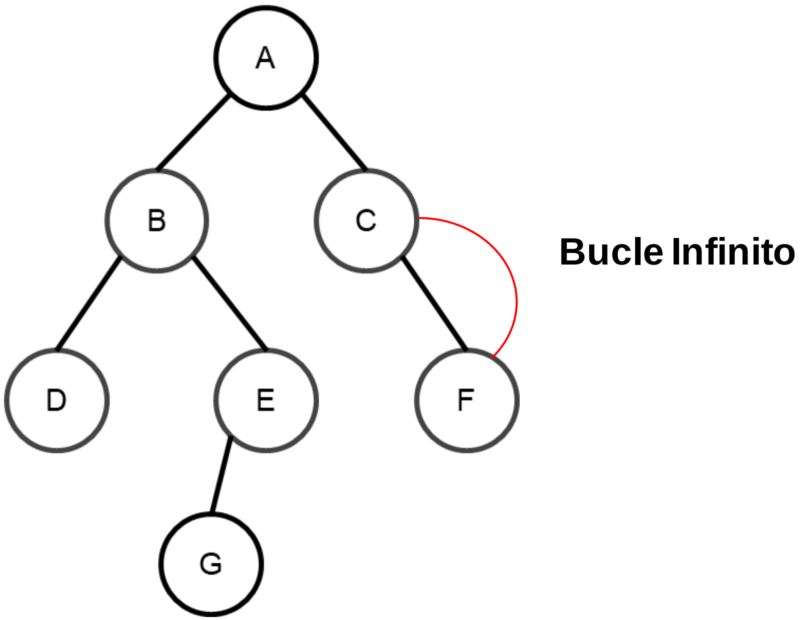
- Cola = {D, E, F}
- Visitados = {A,B,C,D,E,F}




```
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```

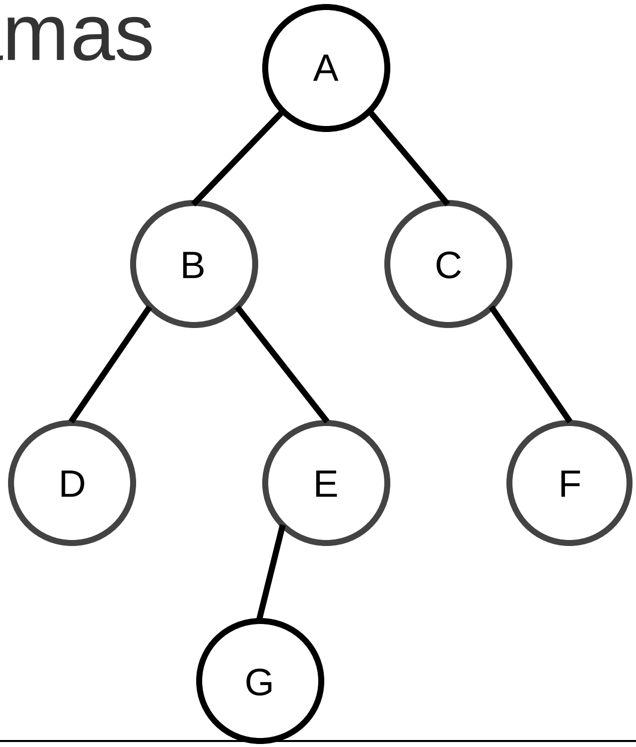
Grafos - BFS

¿Qué utilidad tiene el array de visitados?



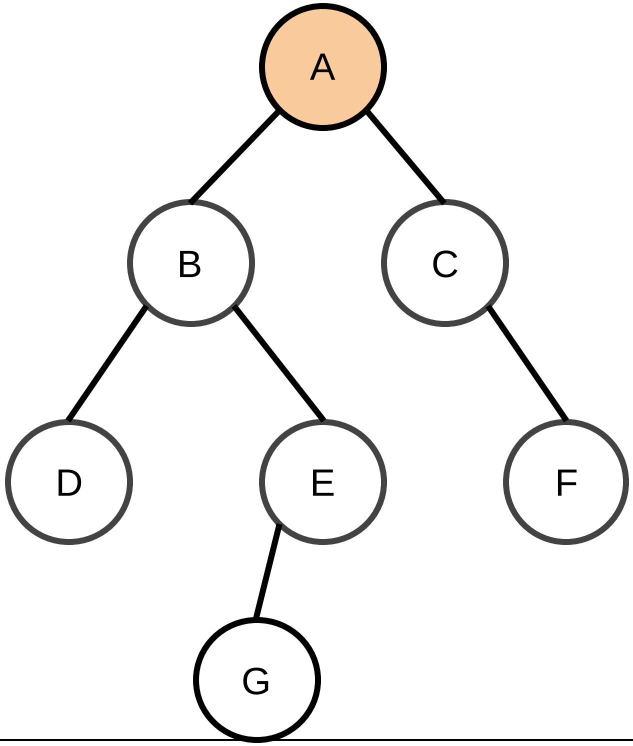
Grafos - DFS

- Recorrido en profundidad
- Equivalente a recorrer ramas



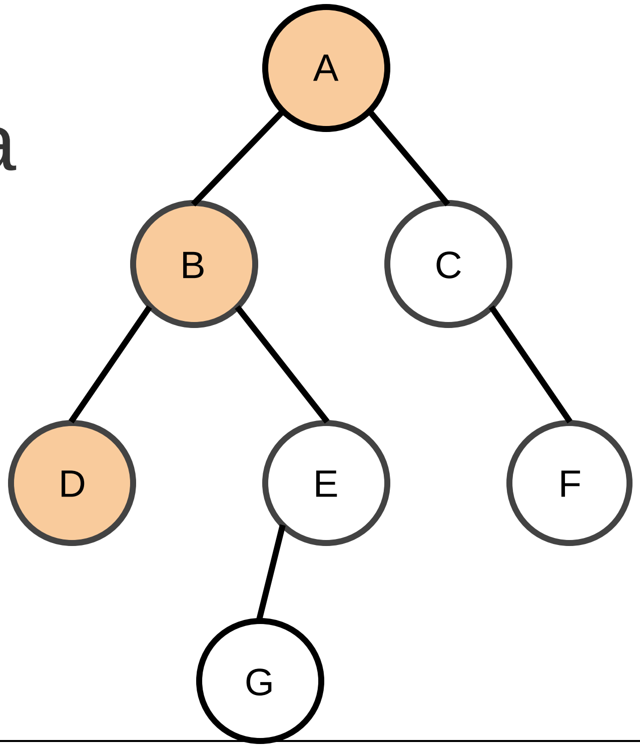
Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A



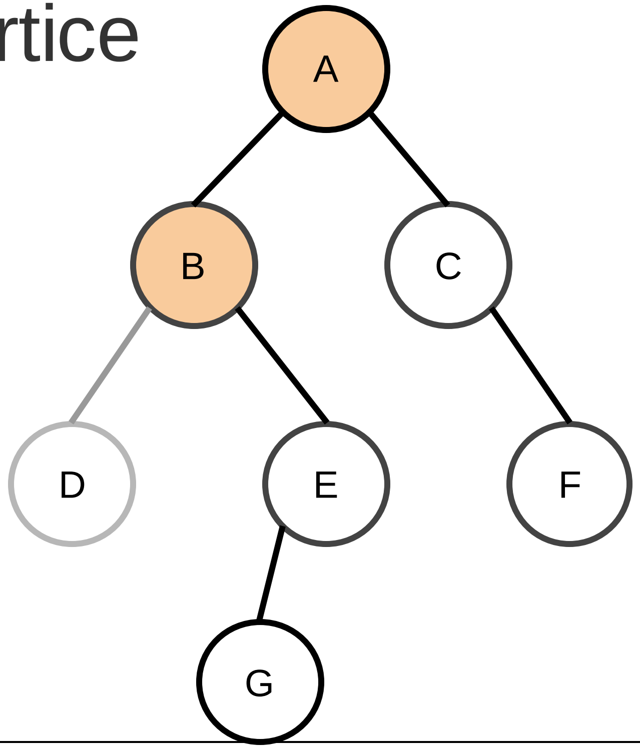
Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A
  - Recorremos una rama hasta el final



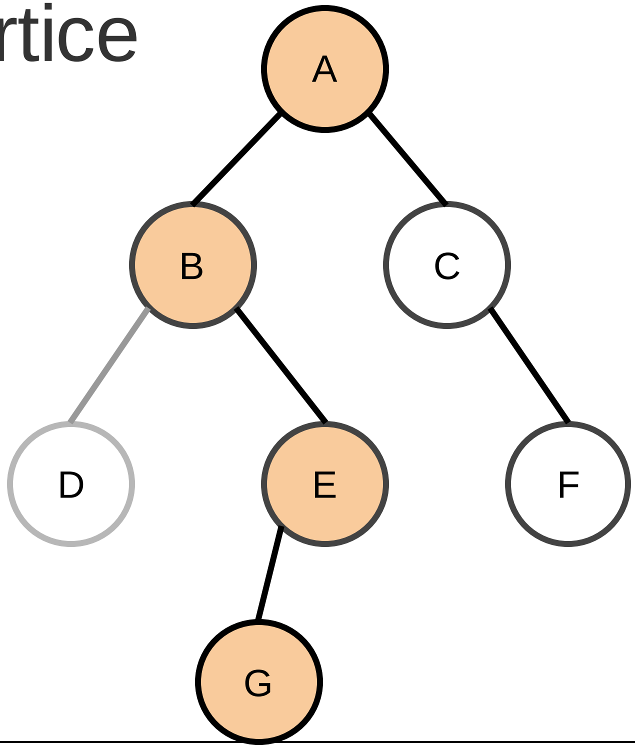
Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (B)



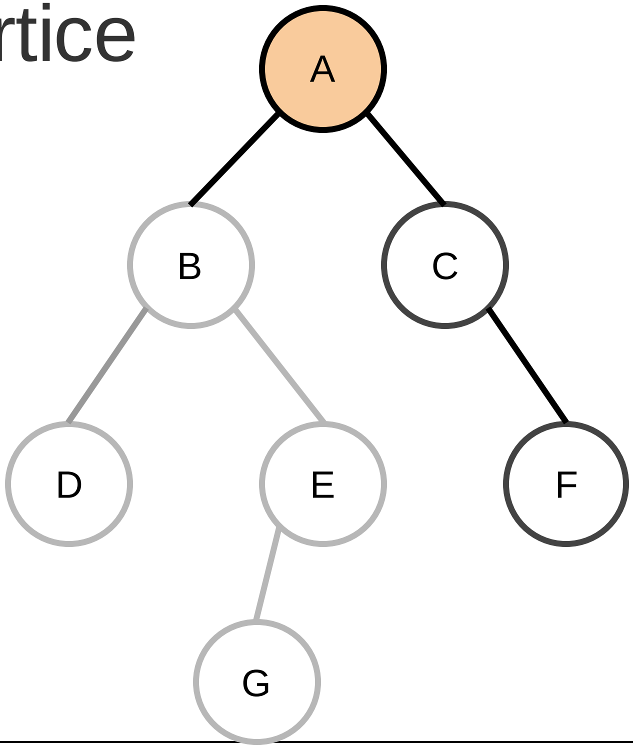
Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (B)
  - recorremos la nueva rama hasta el final



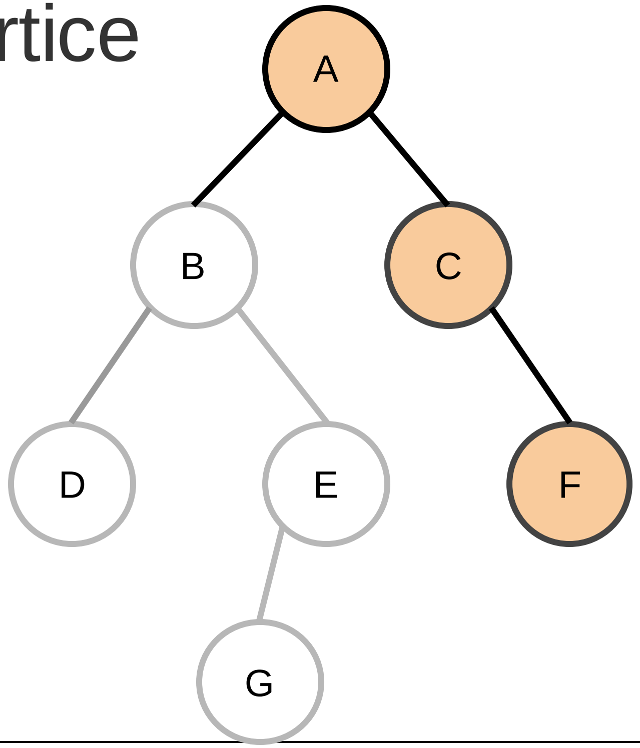
Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (A)



Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (A)



Grafos - DFS

- Implementación

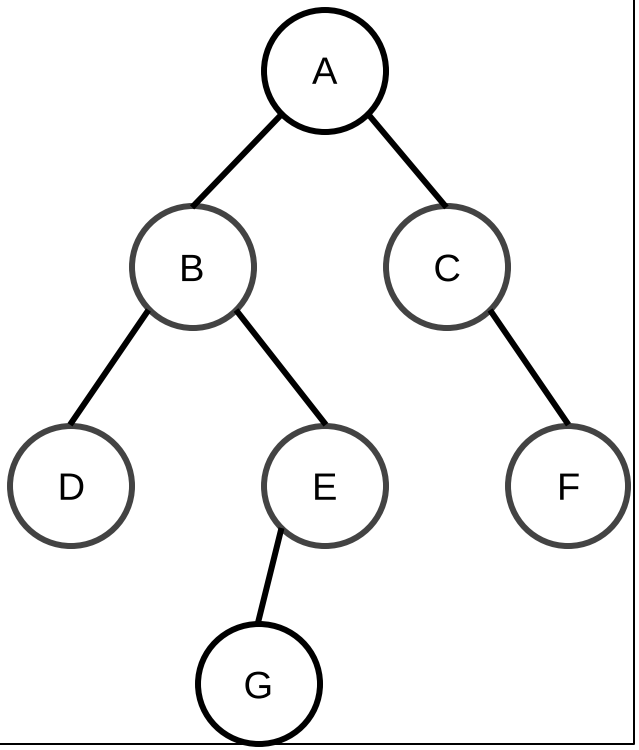
    - Array de valores booleanos (**visitados**)
        - Imprescindible para evitar TLE

    - **Pila** de vértices a explorar
        - Se procesa el más reciente primero

Grafos - DFS

- Inicialización: Elegimos un vértice inicial

```
inicial = A
visitado[inicial]=true
pila.add(inicial)
```



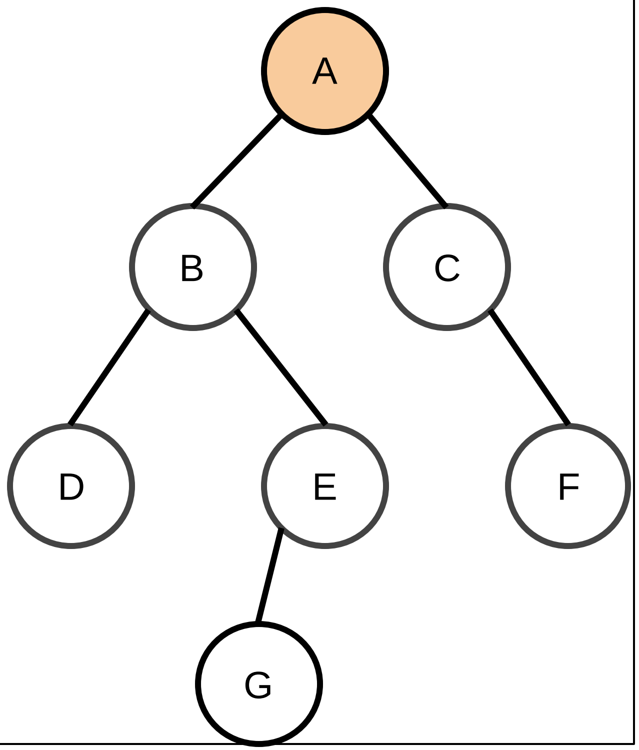
Grafos - DFS

- Pila = {A} ⇐ cima de la pila por la derecha

inicial = A

visitado[inicial]=true

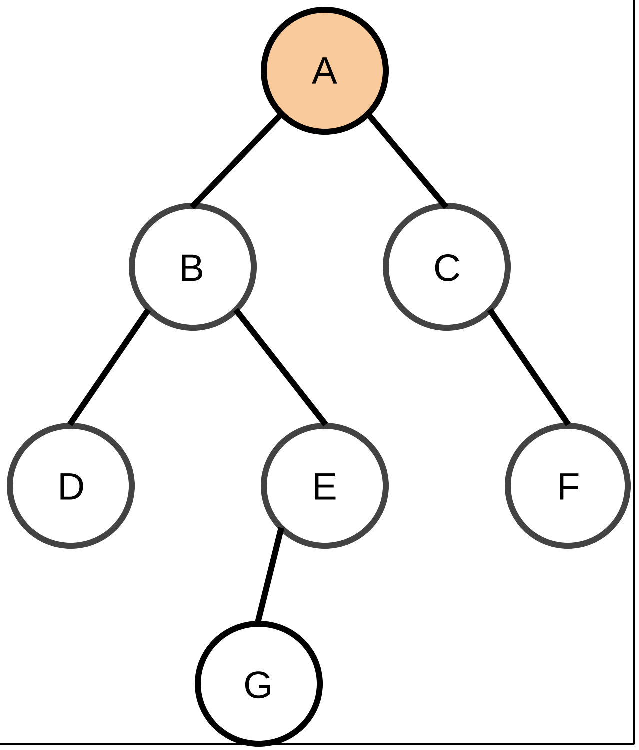
pila.add(inicial)



Grafos - DFS

- Recorremos los vértices

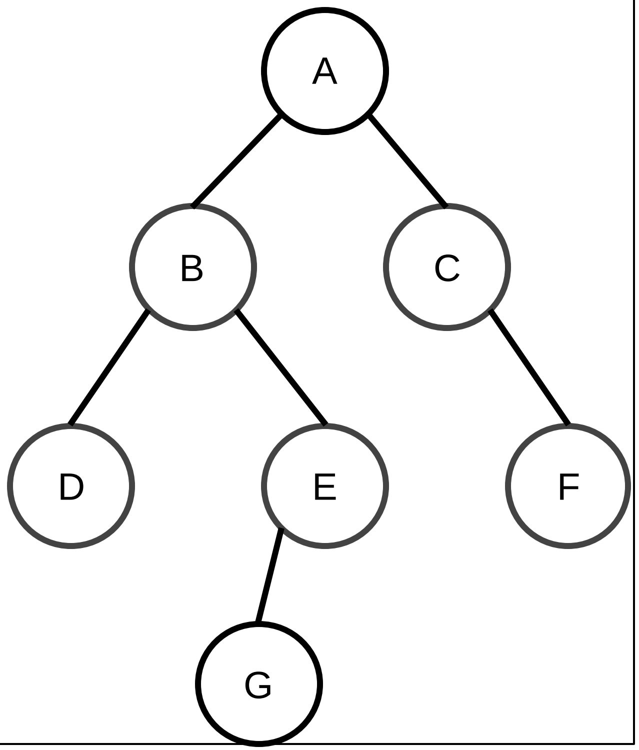
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {}
- Visitados = {A}
- Sacamos A

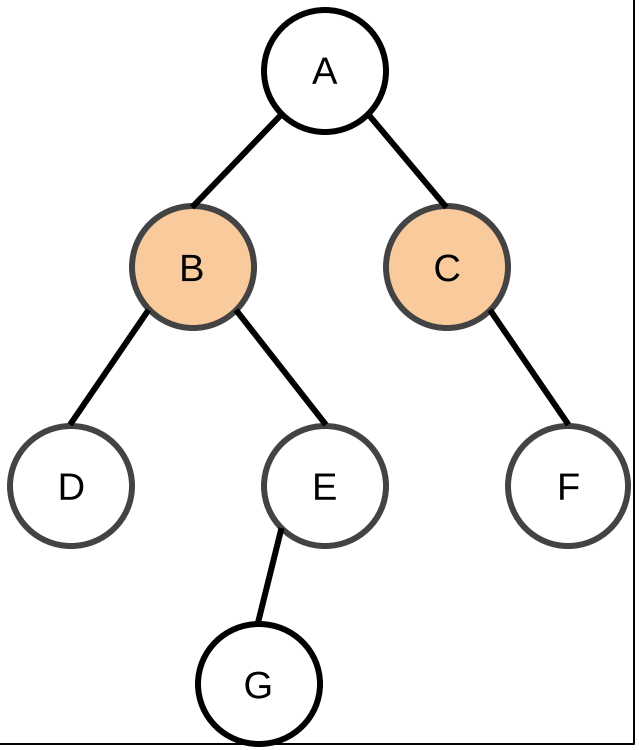
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {B,C}
- Visitados = {A,B,C}

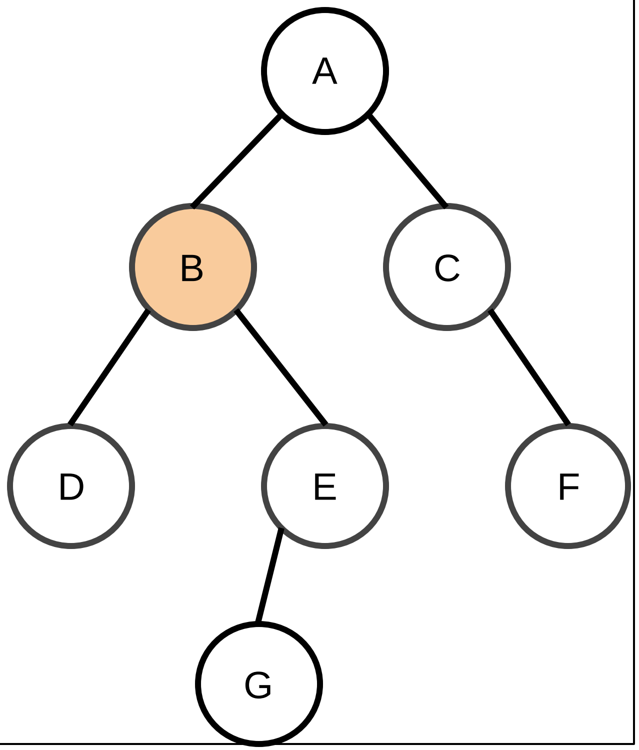
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {B}
- Visitados = {A,B,C}
- Sacar C

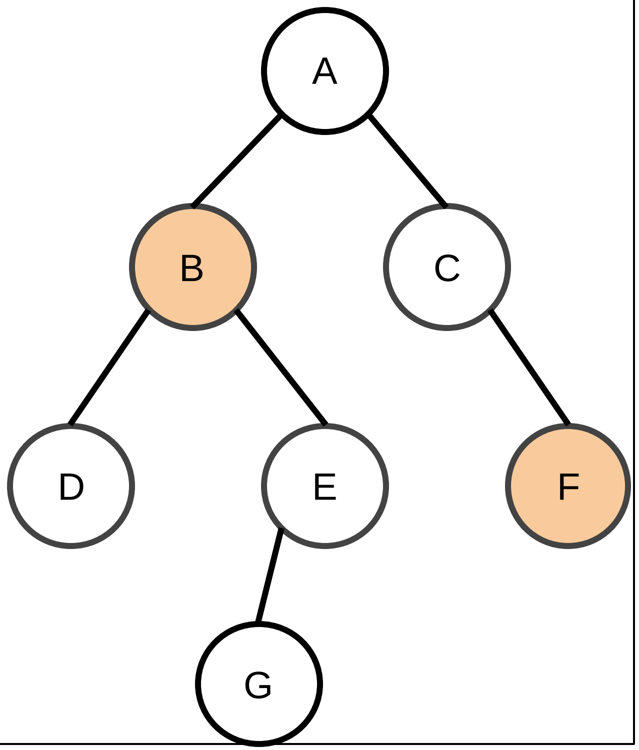
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {B,F}
- Visitados = {A,B,C,F}

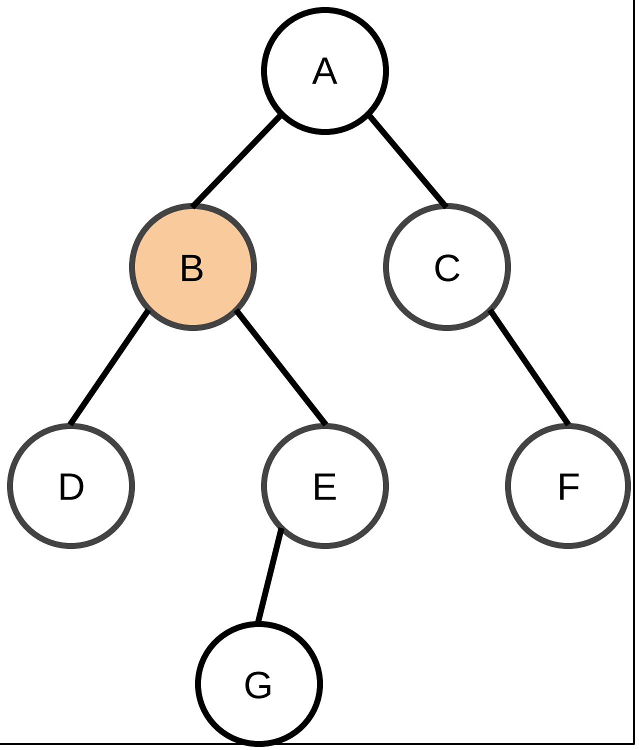
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {B}
- Visitados = {A,B,C,F}
- Sacar F

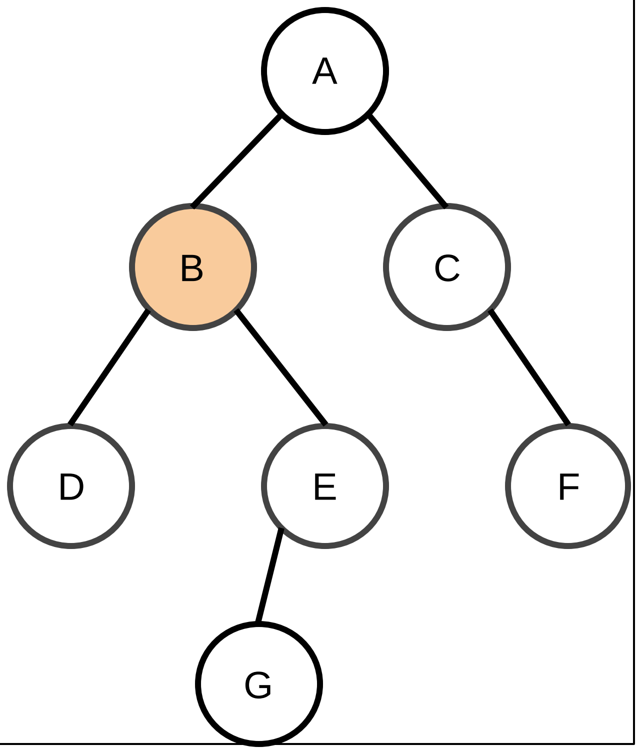
```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

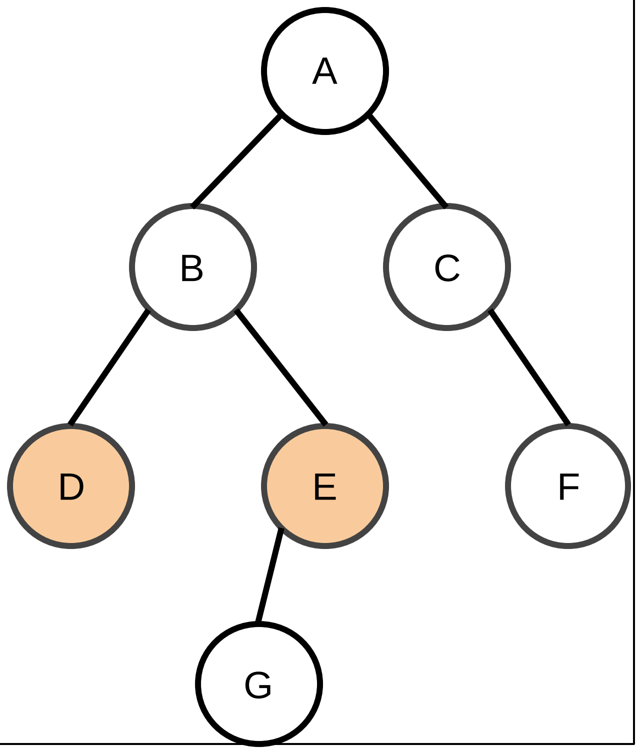
- Pila = {}
- Visitados = {A,B,C,F}
- Sacar B

```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```



Grafos - DFS

- Pila = {D,E}
- Visitados = {A,B,C,F,D,E}




```
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```

Grafos – Implementación BFS/DFS

BFS

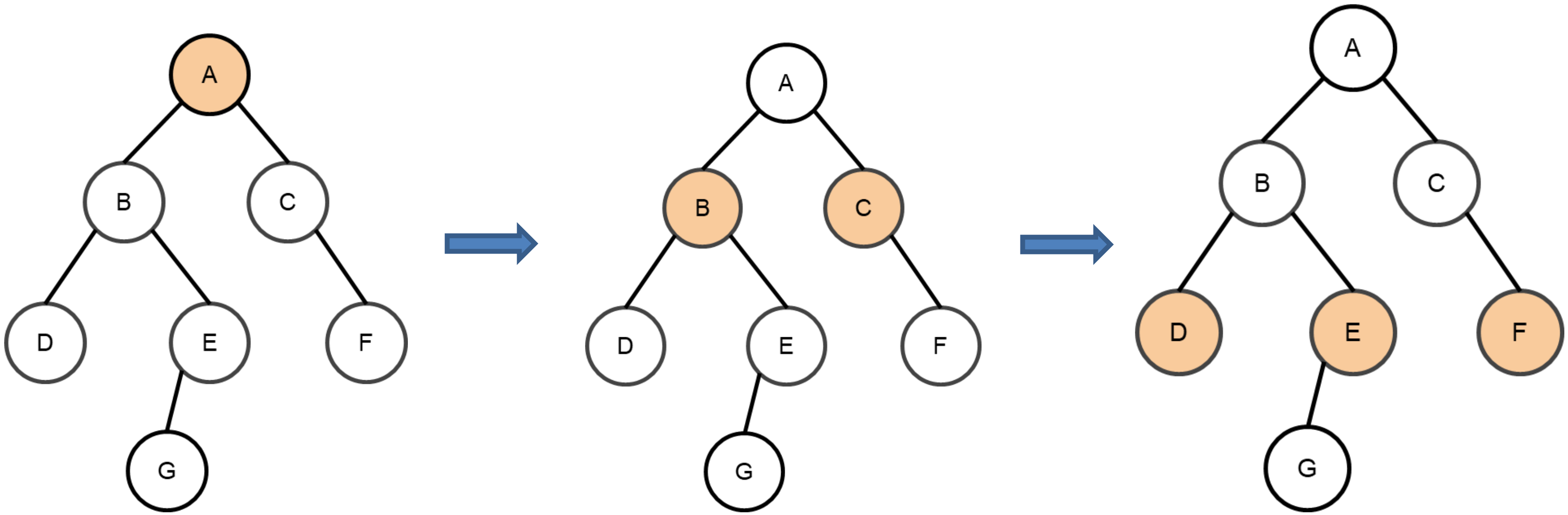
```
inicial = A
visitado[inicial]=true
cola.add(inicial)
mientras(cola.size() > 0)
        v = cola.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```

DFS

```
inicial = A
visitado[inicial]=true
pila.add(inicial)
mientras(pila.size() > 0)
        v = pila.extraer()
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                pila.add(ady)
```

Grafos – BFS Extra

Hemos visto que el **BFS** es un **recorrido por niveles**:



Grafos – BFS Extra

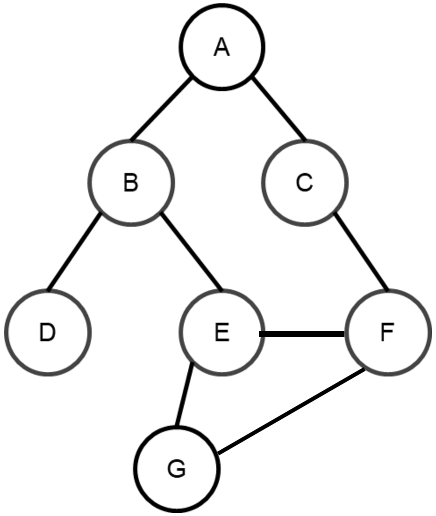
¿Esto nos **puede servir para algo**? **Sí**

¿Cuál es el camino más corto entre el nodo A y F?

Opción 1: {A-B-E-F}

Opción 2: {A-B-E-G-F}

Opción 3: {A-C-F}



Solución: {A-C-F}

Grafos – BFS Extra

¿**Qué hemos hecho** para encontrar ese camino?

-Recorrer el grafo por niveles.

Luego el recorrido con **BFS nos va a permitir calcular el camino más corto entre dos nodos** en un grafo no ponderado

Grafos – BFS Extra

Una implementación para contar los niveles o la distancia de un nodo a otro es la siguiente:

```
inicial = A
visitado[inicial]=true
cola.add(inicial)
cola.add(-1)
mientras(pila.size() > 1)
        v = cola.extraer()
        if(v == -1)
                niveles++
                cola.add(-1)
        para cada ady de v:
        if(no visitado[ady])
                visitado[ady]=true
                cola.add(ady)
```

Evitar bucle infinito con solo -1s

Grafos – DFS Extra

- También se puede utilizar un DFS recursivo para recorrer el grafo:

```
DFS(v)
        visitado[v] = true
        para cada ady de v:
        si (no visitado[ady])
                DFS(ady)
```

Grafos – Resumen BFS/DFS

BFS

-**Cola**

-**Propiedad**: Camino más corto de un nodo a otro en grafo no ponderado

- Complejidad: O(V + E)

DFS

-**Pila**

-Implementación alternativa con recursión

-Complejidad: O(V + E)

CUIDADO EN PYTHON CON RECURSION LIMIT!

Grafos - Componentes Conexas

- **¿Qué es una componente conexa?**

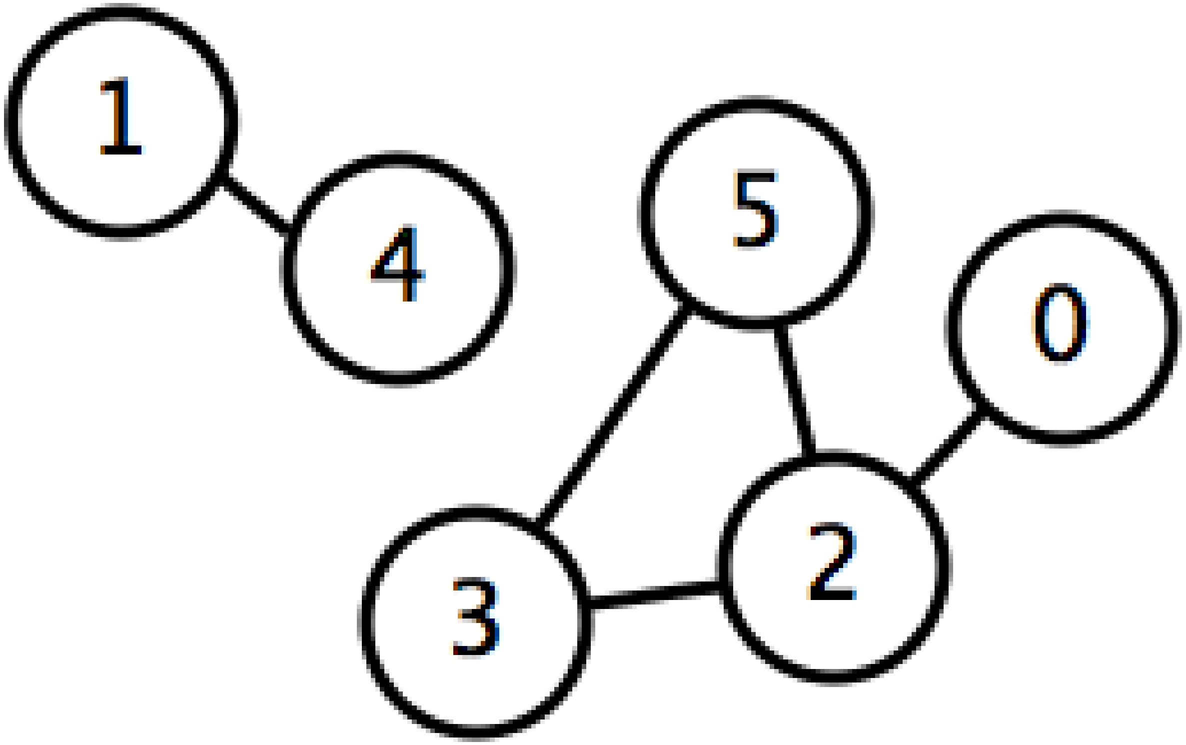
Es un subgrafo donde dos vértices cualesquiera están conectados a través de uno o más caminos

Grafos | Componentes Conexas

- La idea básica es hacer un BFS/DFS por cada vértice i desde 0..N siempre y cuando i no haya sido recorrido por un recorrido anterior

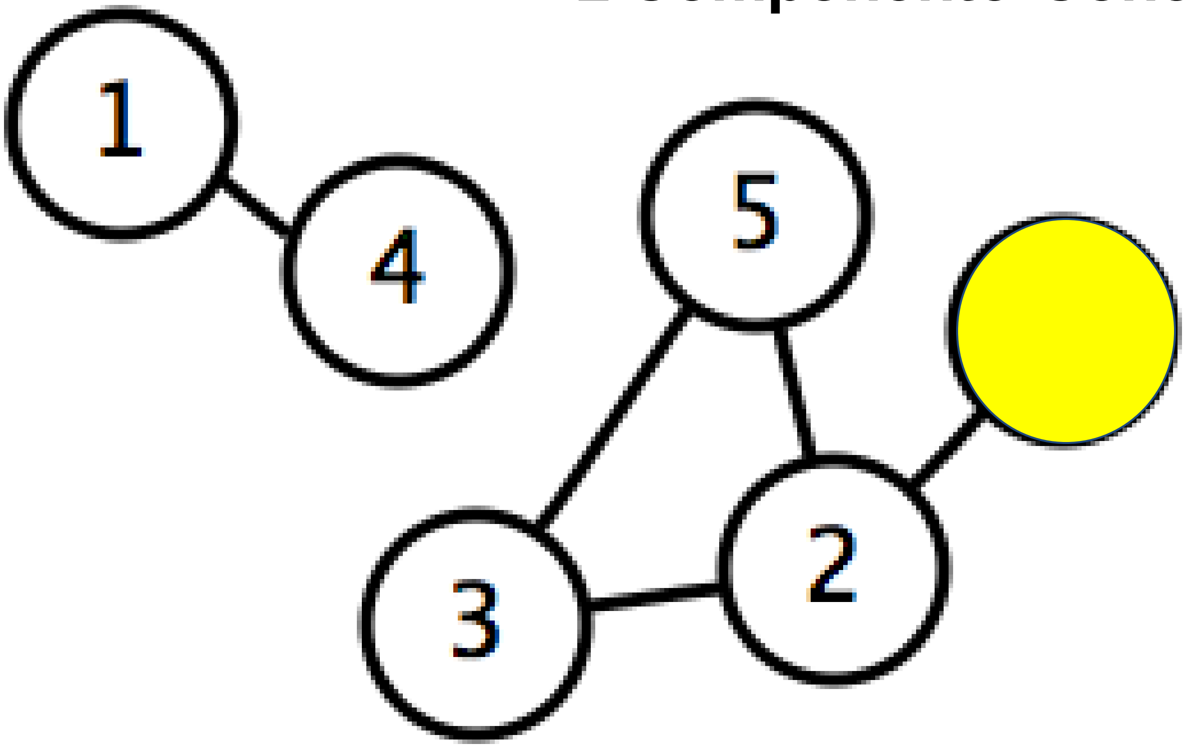
- Se cuenta 1 y se recorre i

Grafos | Componentes Conexas

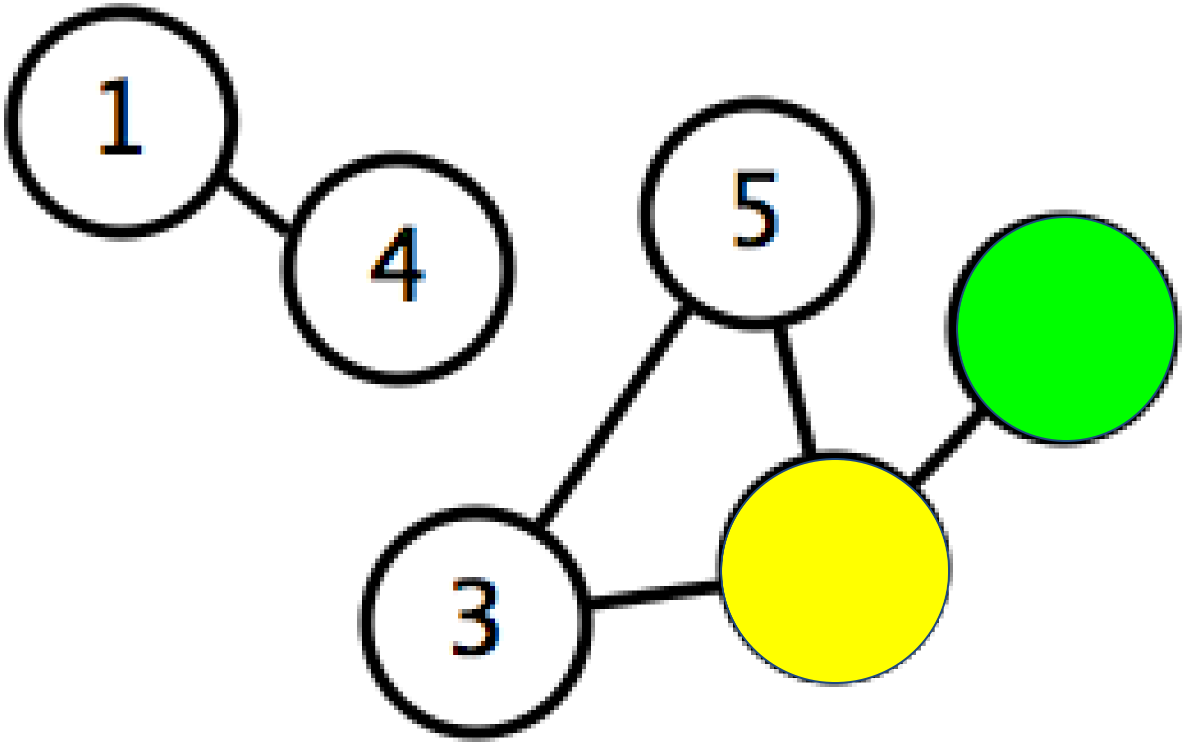


Grafos | Componentes Conexas

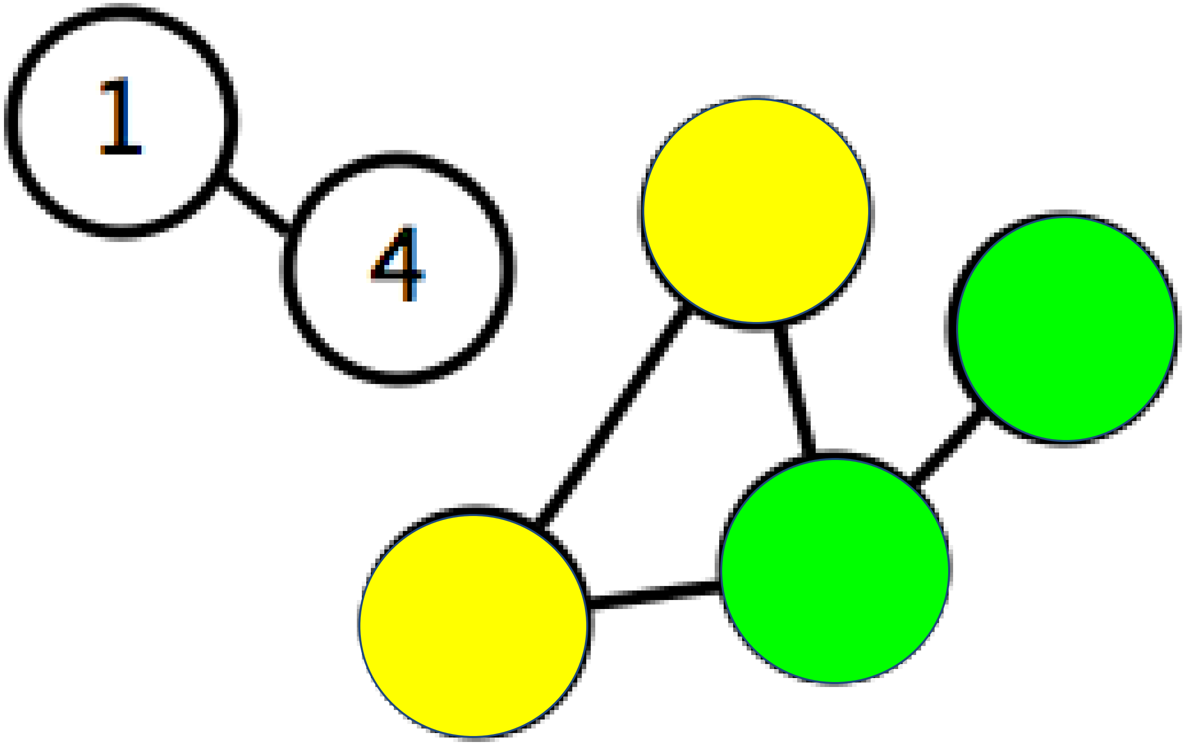
+1 Componente Conexa (1)



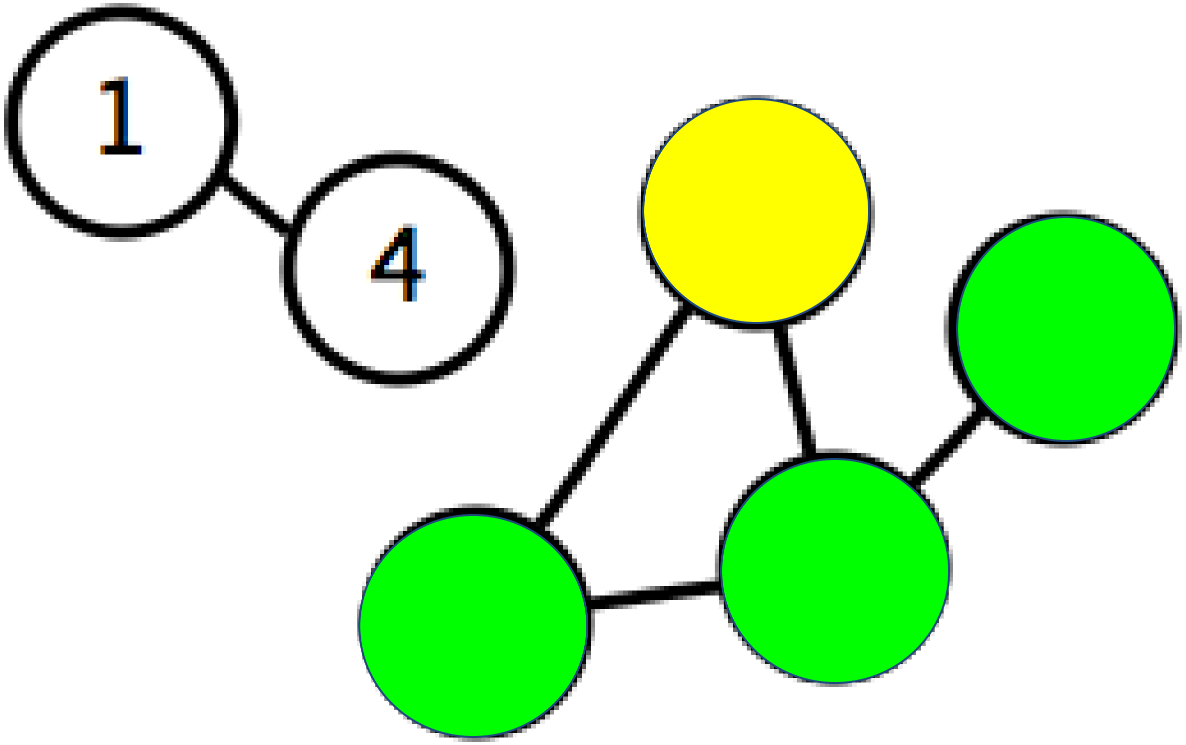
Grafos | Componentes Conexas



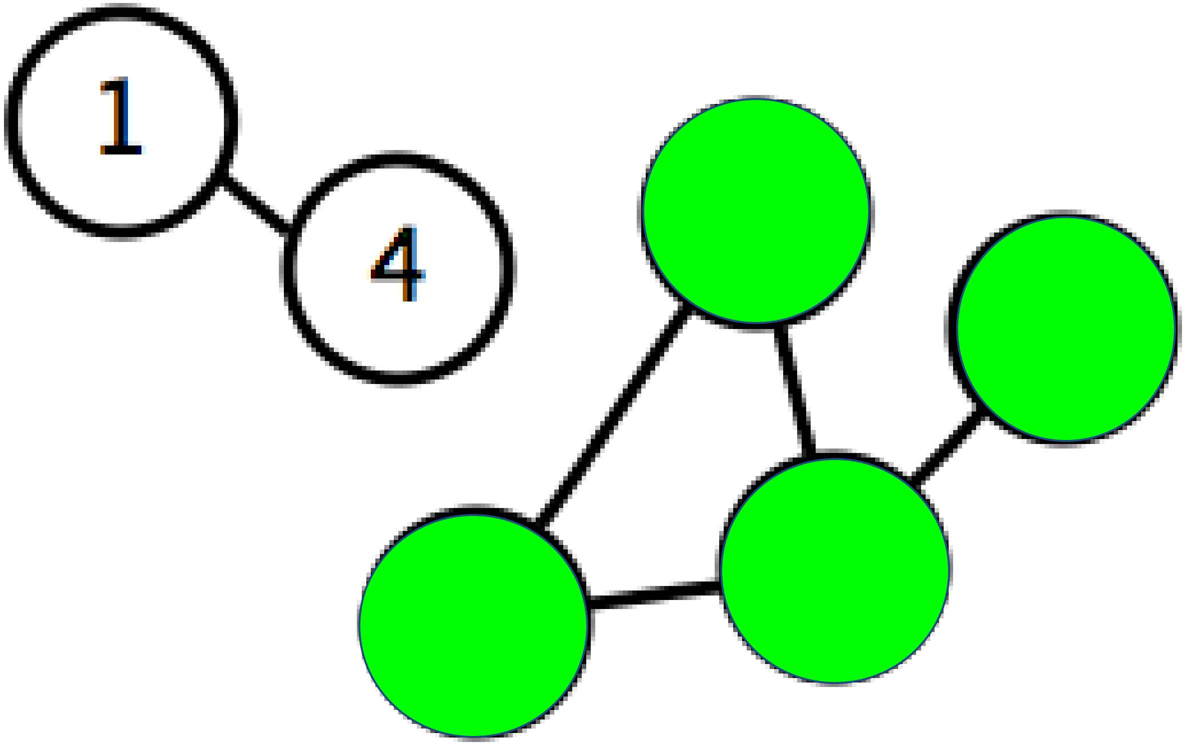
Grafos | Componentes Conexas



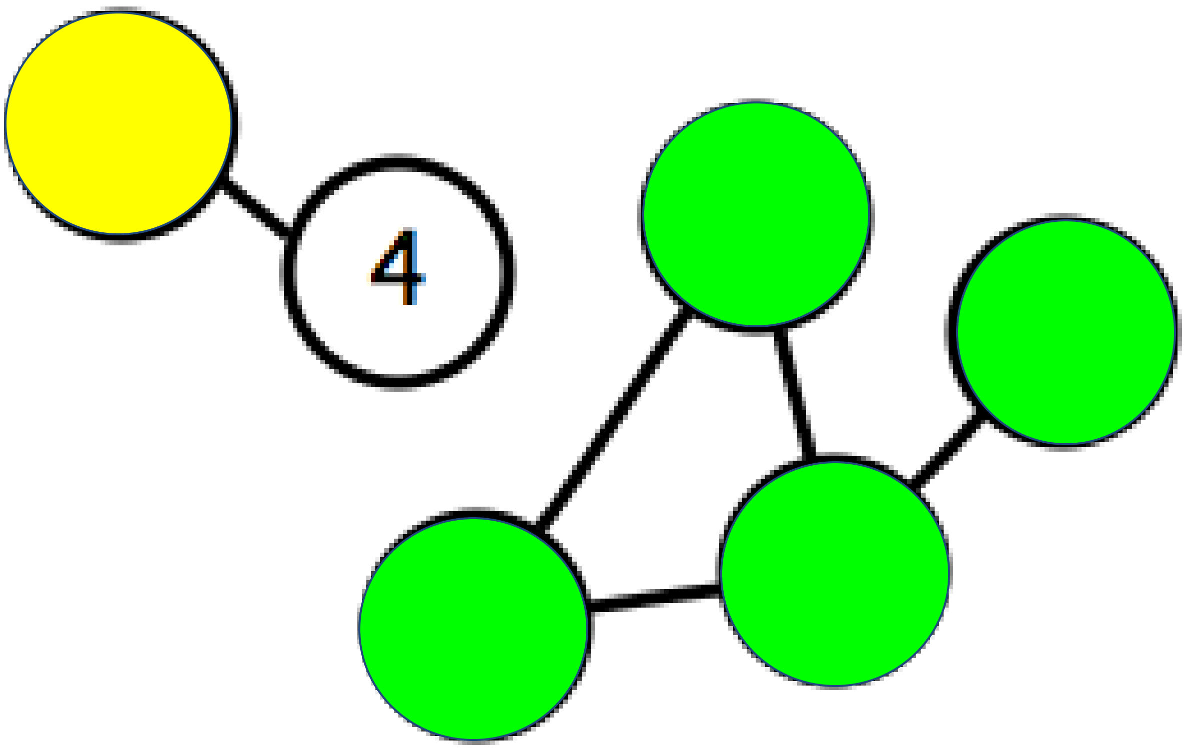
Grafos | Componentes Conexas



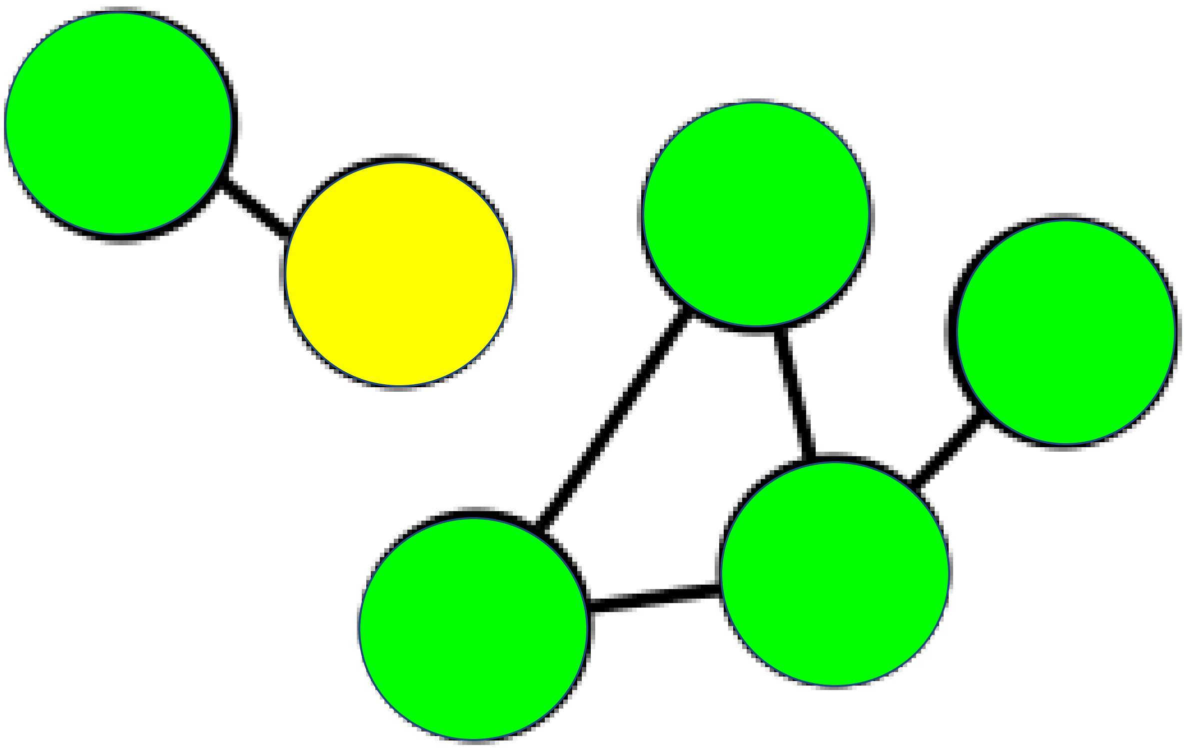
Grafos | Componentes Conexas



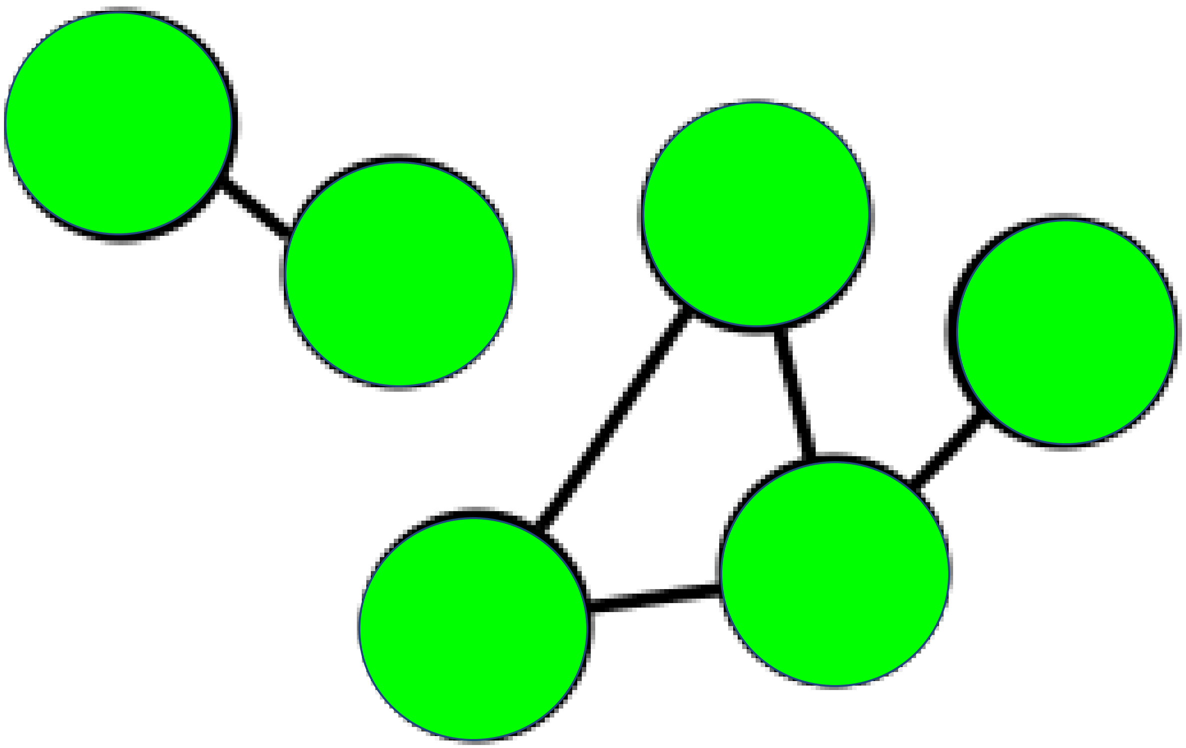
+1 Componente Conexa (2)



Grafos | Componentes Conexas



Grafos | Componentes Conexas



Grafos | Componentes Conexas

- Pseudocódigo:

```
grafo = vector<int>[n]
visitados = boolean[n]
for(int i=0;i<n;i++)
        if(!visitados[i])
                BFS/DFS(i)
                compoConexas++
```

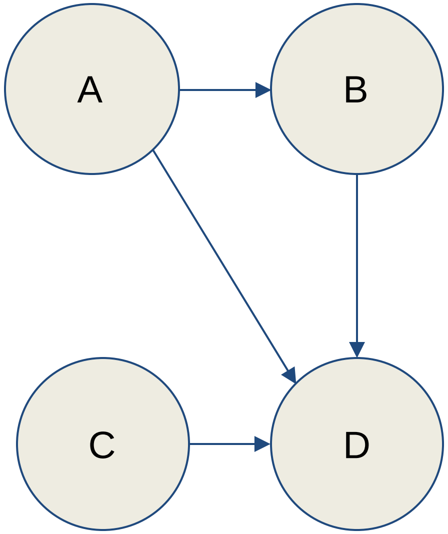
Grafos | Ordenamiento Topológico

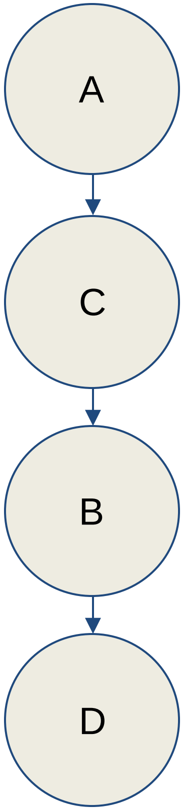
- Sobre un grafo acíclico dirigido (DAG)

- Ordenar nodos tal que para cualquier nodo u,v; al momento de eliminar u, v no contenga aristas hacia ella

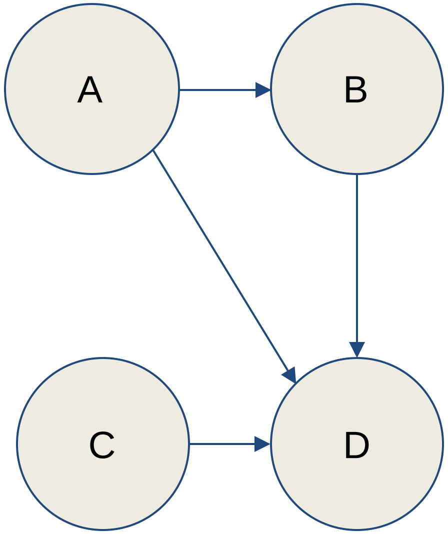
- Utilizar el algoritmo **DFS** (versión recursiva)

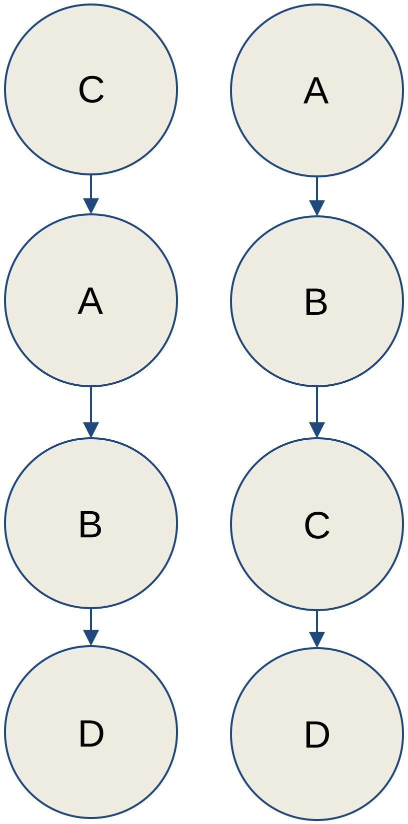
Grafos | Ordenamiento Topológico



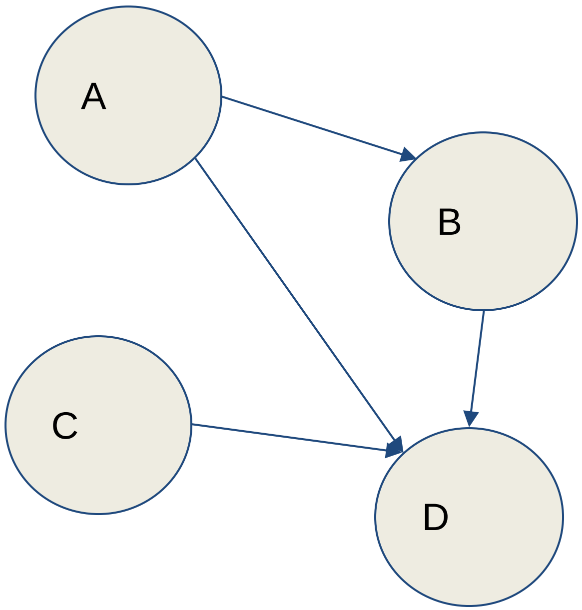


Grafos | Ordenamiento Topológico



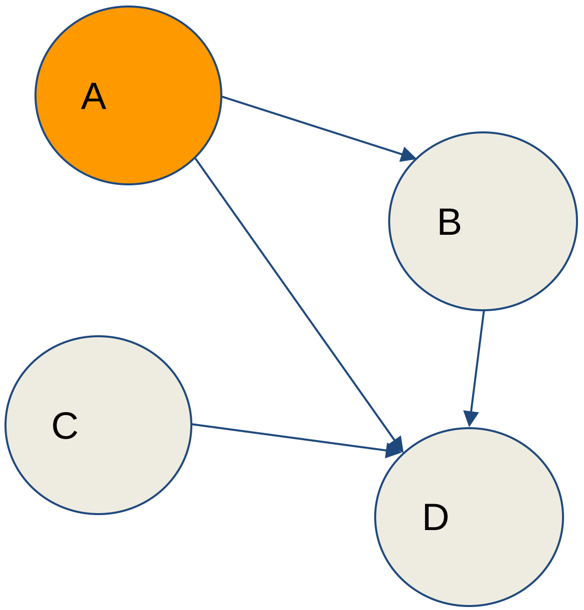


Grafos | Ordenamiento Topológico



A y C no les incide ningún otro nodo

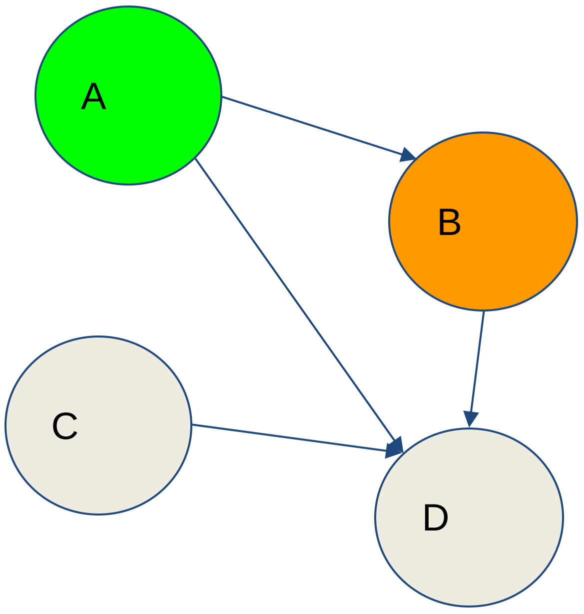
Grafos | Ordenamiento Topológico



TS = { }

DFS(A) = DFS(B), DFS(D)

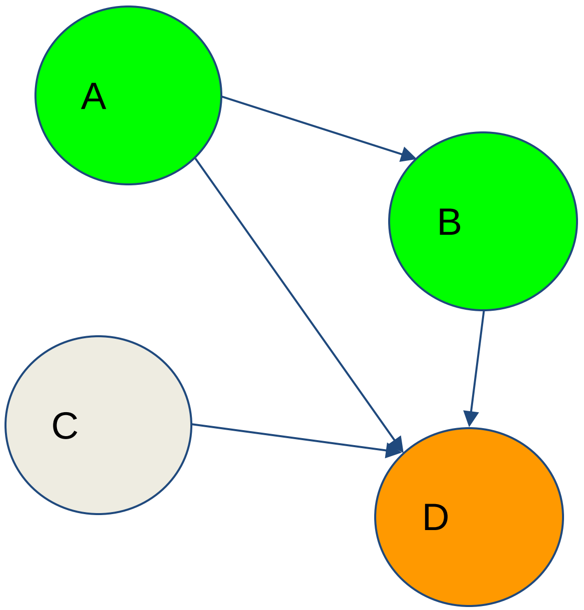
Grafos | Ordenamiento Topológico



TS = { }

DFS(B) = DFS(D)

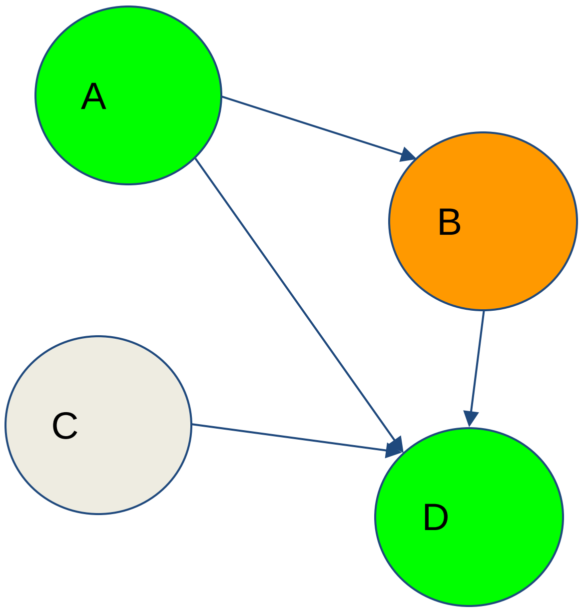
Grafos | Ordenamiento Topológico



TS = {D}

DFS(D) = ø, TS.push(D)

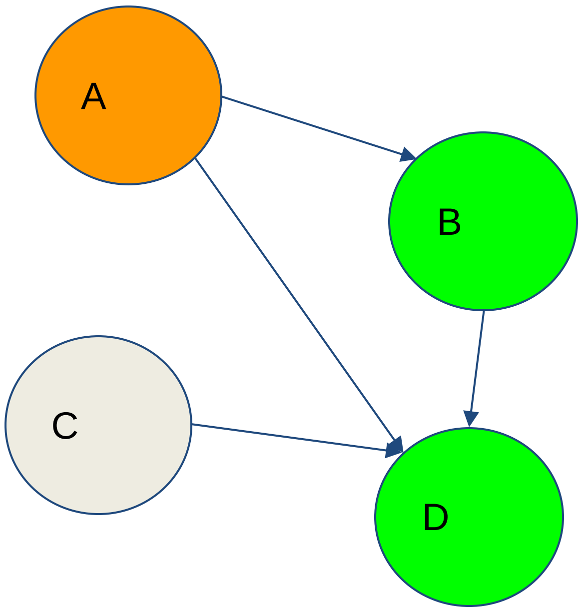
Grafos | Ordenamiento Topológico



TS = { B, D}

DFS(B) = DFS(D), TS.push(B)

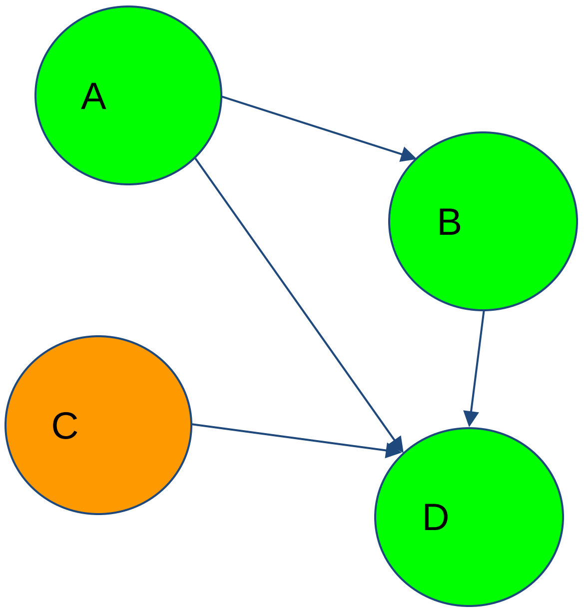
Grafos | Ordenamiento Topológico



TS = { A, B, D}

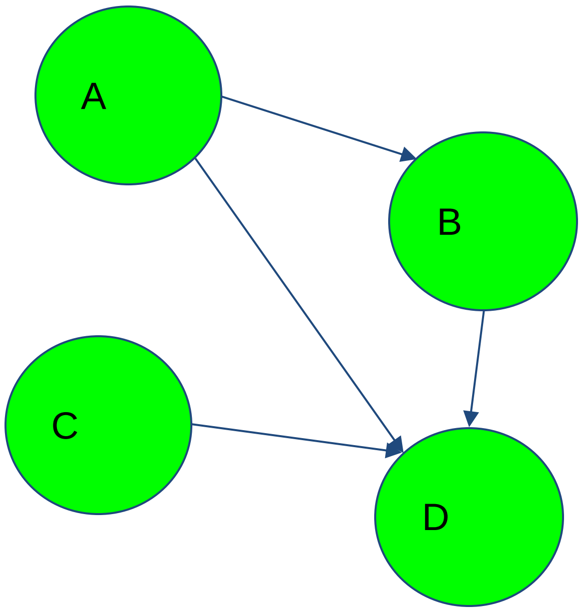
DFS(A) = DFS(B), DFS(D), TS.push(A)

Grafos | Ordenamiento Topológico



TS = { C, A, B, D}

DFS(C) = DFS(D), TS.push(C)



TS = { C, A, B, D}

A y C pueden ir en cualquier orden, por lo que los toposort son => {C,A,B,D} ó {A,C,B,D}

Grafos | Ordenamiento Topológico

- Pseudocódigo

```
TopoSort(n):
    v[n] = true
    for edge in edges(n):
        TopoSort(edge)
    pila.push(n)
```

Grafos | Ordenamiento Topológico

- Se almacenan los nodos en una **pila**
- Siendo **edges(n) una lista con los vértices** con los que n se conecta (N → A)
- Siendo **V(N) un array de booleanos** donde se entiende que si V(N) = true ya ha pasado un recorrido en profundidad por el nodo N

Grafos | Puntos de Articulación

- Dado un grafo, nos interesa saber que nodos, de ser eliminados, hacen que el número de componentes conexas del grafo incremente
- **Idea ingenua**: Contar componentes conexas "haciendo como si no existe" cualquier vértice u del grafo G={V,E}

Grafos | Puntos de Articulación

- Esta idea es $O(V * (V + E))$ con V = nodos del grafo.

- Luego $O(V^2 + VE) = O(V^2)$

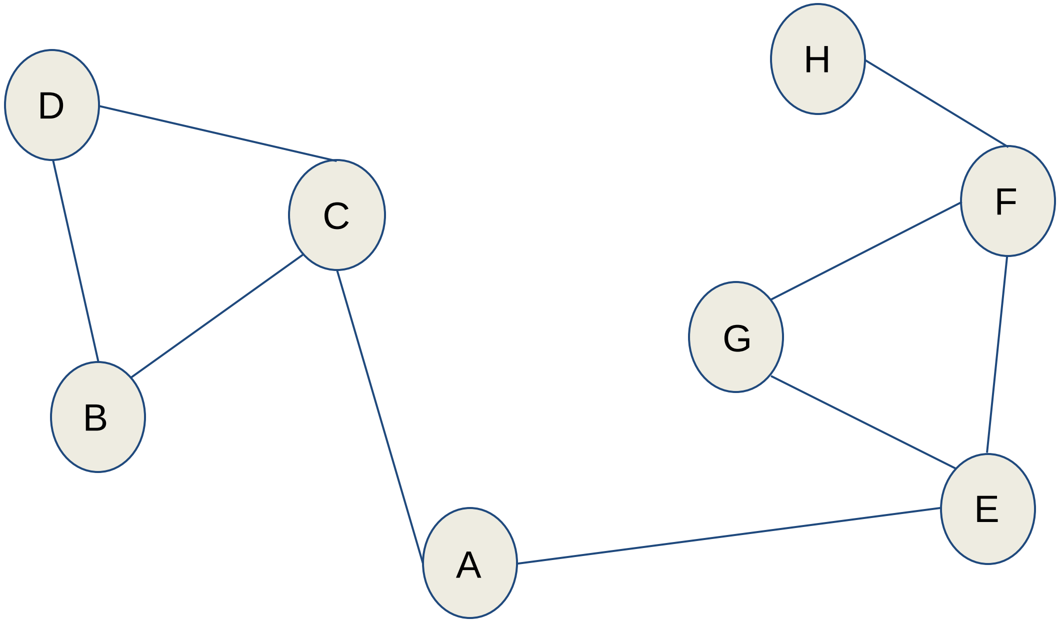
Grafos | Puntos de Articulación

- **Algoritmo de Tarjan**
  - Tomamos un nodo cualquiera y recorremos a través de **DFS** ese nodo
  - Contamos el tiempo en el que llegamos a cualquier nodo “v” (visitar 1 nodo suma 1 al tiempo). Llamamos a esto, “**discovery**”
  - Con esta misma idea podemos observar si se puede llegar a cualquier nodo por otro “camino”. Llamamos a esto, “**lowest**”

Grafos | Puntos de Articulación

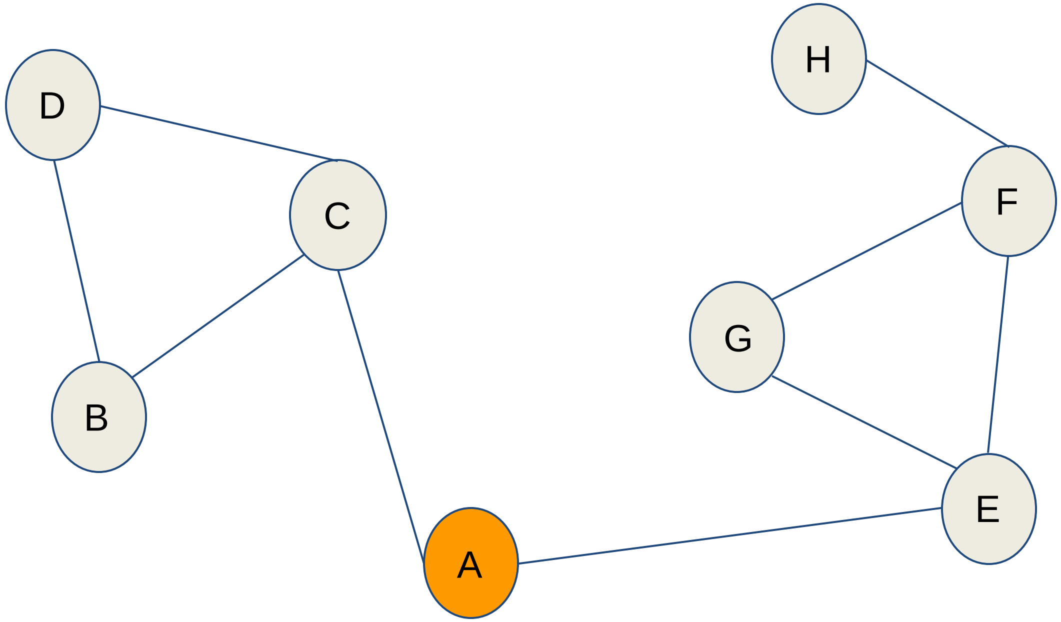
- Algoritmo de Tarjan
  - Condiciones para **punto de articulación** en u:
    i. El **nodo raíz** u (inicio) hizo **más de un DFS** hacia sus vecinos
    ii. Cualquier **nodo no-raíz** v conectado a u tiene un mínimo de tiempo (**lowest) mayor o igual al** tiempo que se descubrió (**discover**) un nodo u

Grafos | Puntos de Articulación



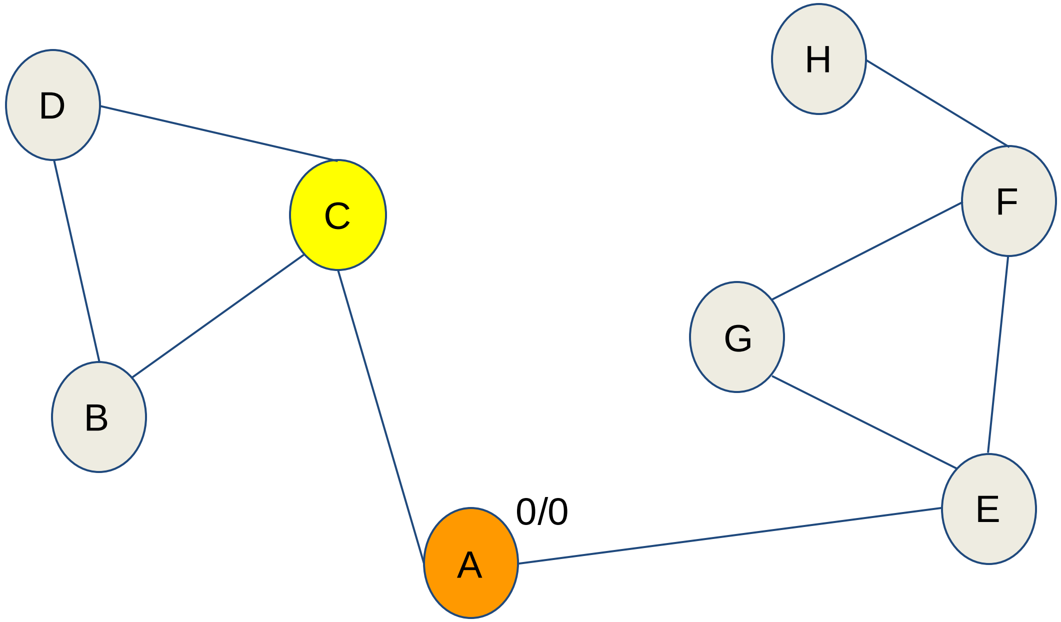
DFS

Grafos | Puntos de Articulación



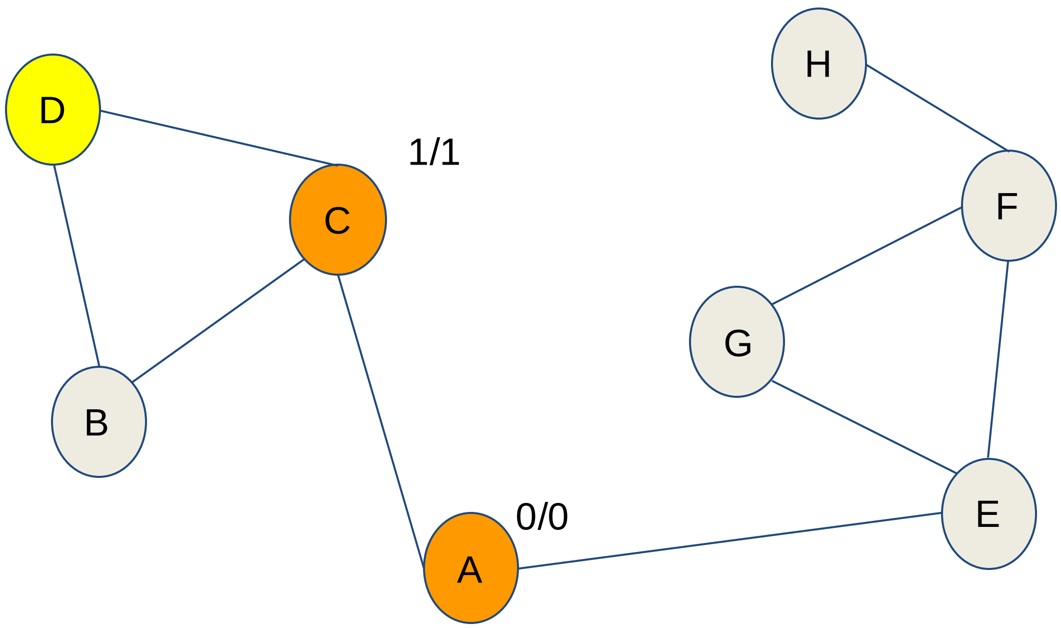
DFS_STACK = (A, 1)
LOWEST = { 0, -1, -1, -1, -1, -1, -1, -1 }
DISCOVERY = { 0, -1, -1, -1, -1, -1, -1, -1 }
VISITED = {1, 0, 0, 0, 0, 0, 0, 0}
PARENTS = { -1, -1, -1, -1, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



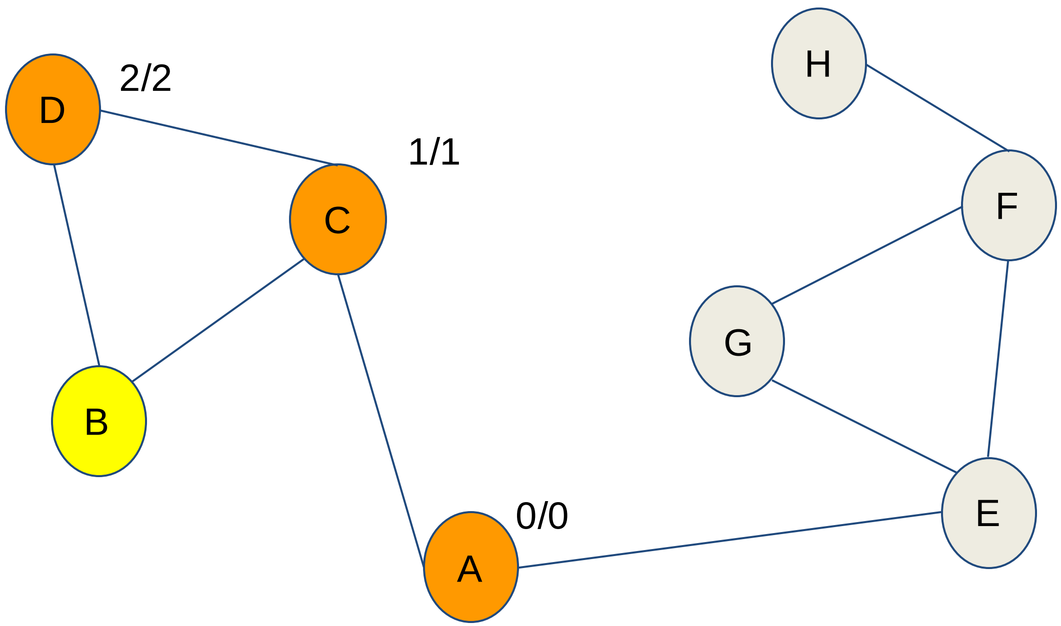
DFS_STACK = (C, 2)
LOWEST = { 0, -1, 1, -1, -1, -1, -1, -1 }
DISCOVERY = { 0, -1, 1, -1, -1, -1, -1, -1 }
VISITED = {1, 0, 1, 0, 0, 0, 0, 0}
PARENTS = { -1, -1, 0, -1, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



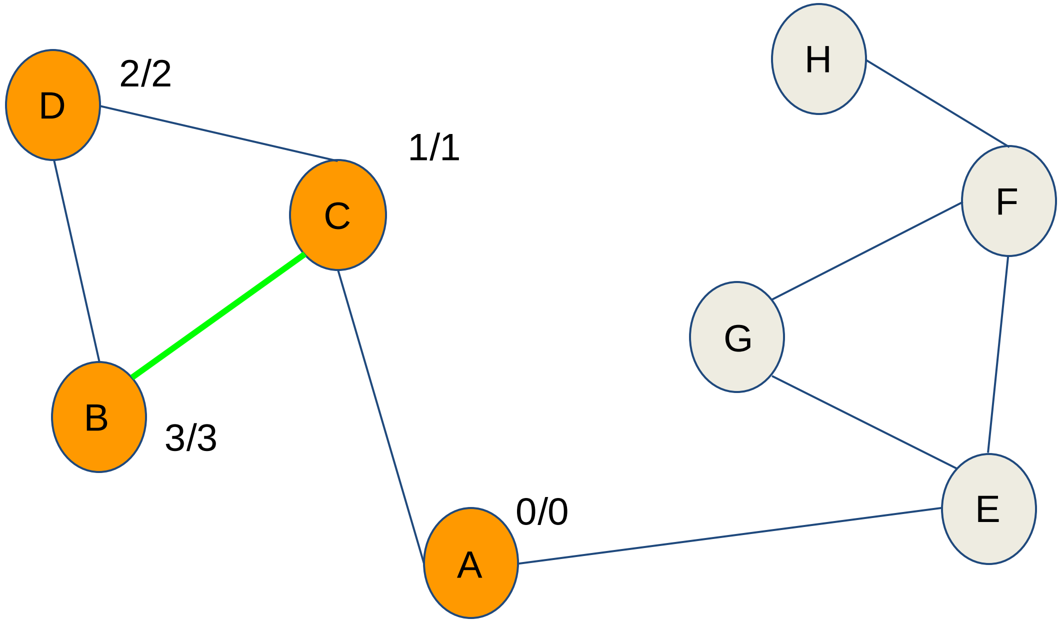
DFS_STACK = (D, 3)
LOWEST = { 0, -1, 1, 2, -1, -1, -1, -1 }
DISCOVERY = { 0, -1, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 0, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, -1, 0, 2, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



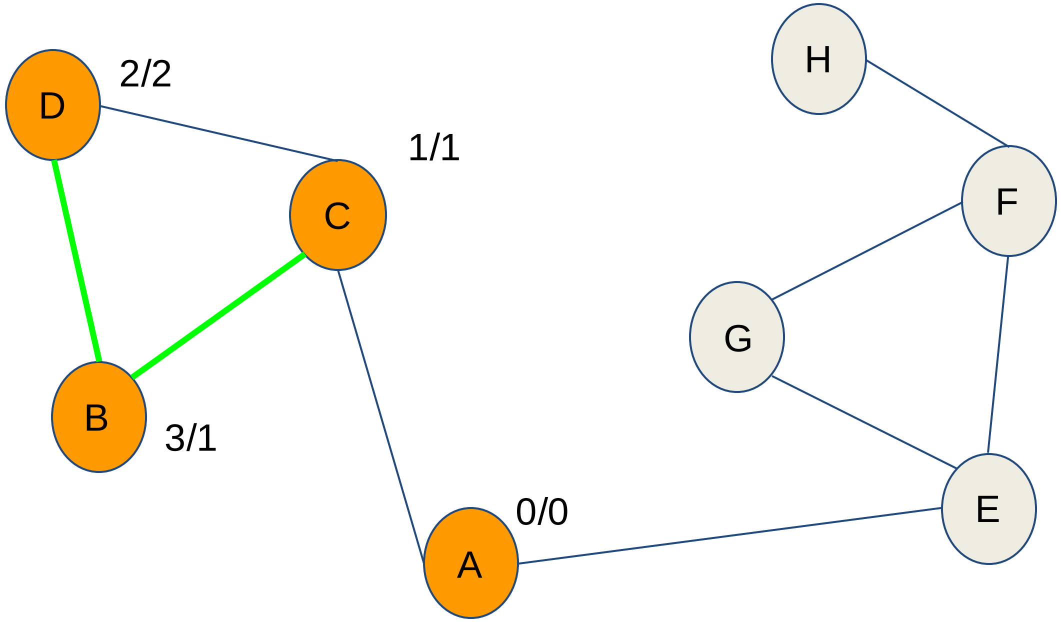
DFS_STACK = (B, 4)
LOWEST = { 0, 3, 1, 2, -1, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



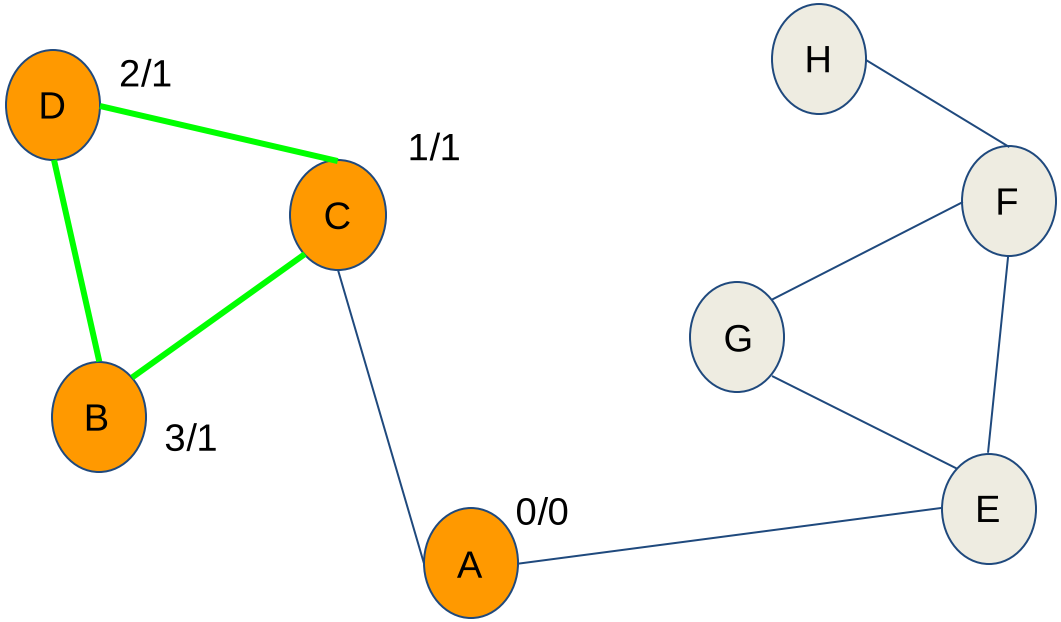
DFS_STACK = (B, 4)
LOWEST = { 0, 3, 1, 2, -1, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



DFS_STACK = (B, 4)
LOWEST = { 0, 1, 1, 2, -1, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

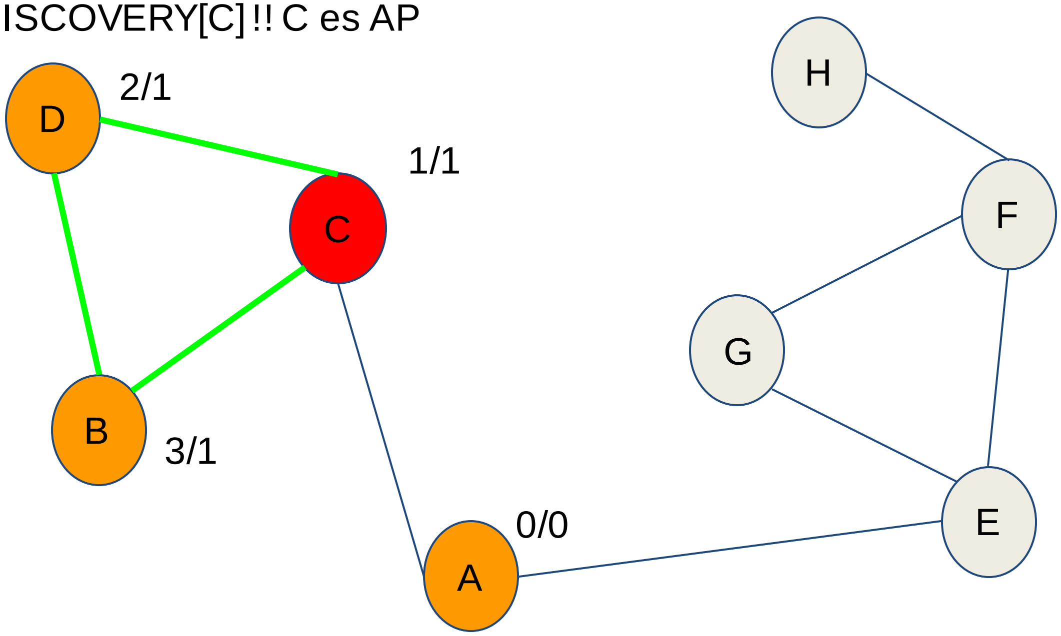
Grafos | Puntos de Articulación



DFS_STACK = (B, 4)
LOWEST = { 0, 1, 1, 1, -1, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

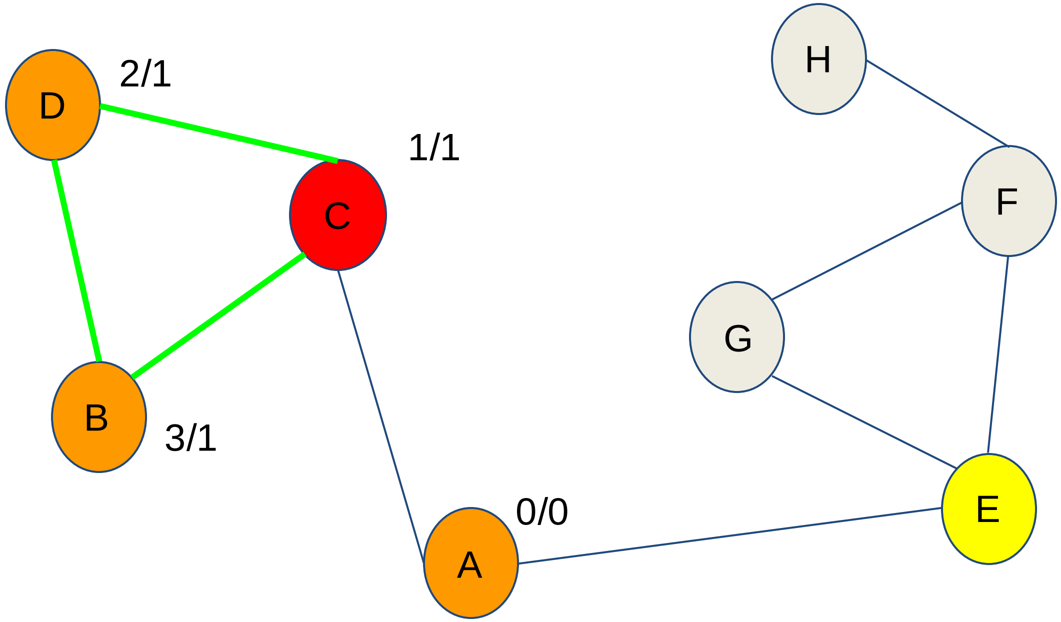
Grafos | Puntos de Articulación

LOWEST[D] >= DISCOVERY[C] !! C es AP



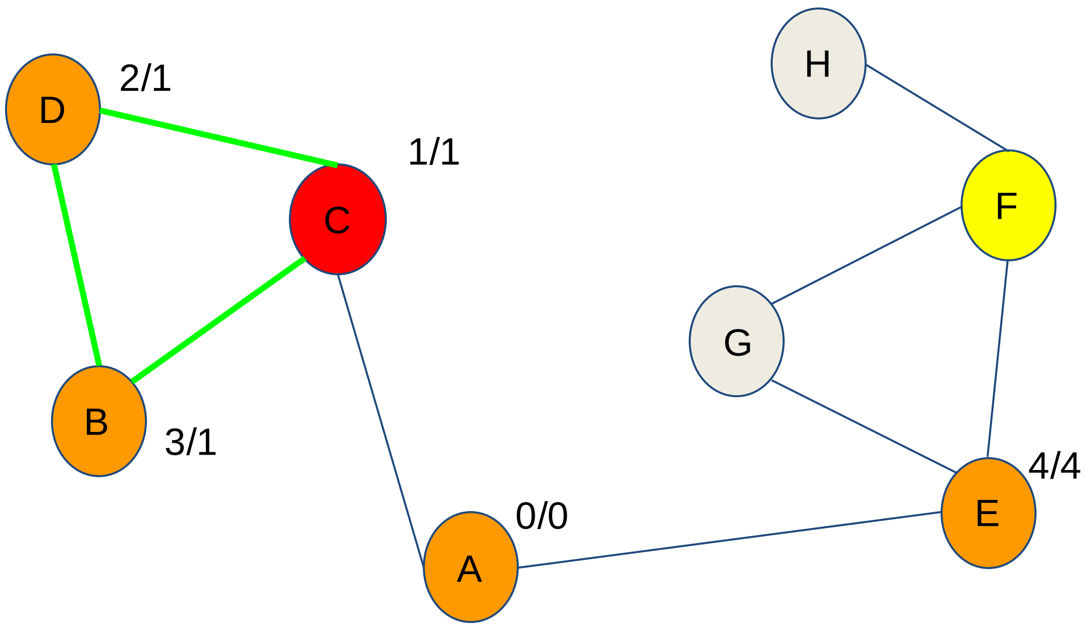
DFS_STACK = (B, 4)
LOWEST = { 0, 1, 1, 1, -1, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 0, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

Grafos | Puntos de Articulación



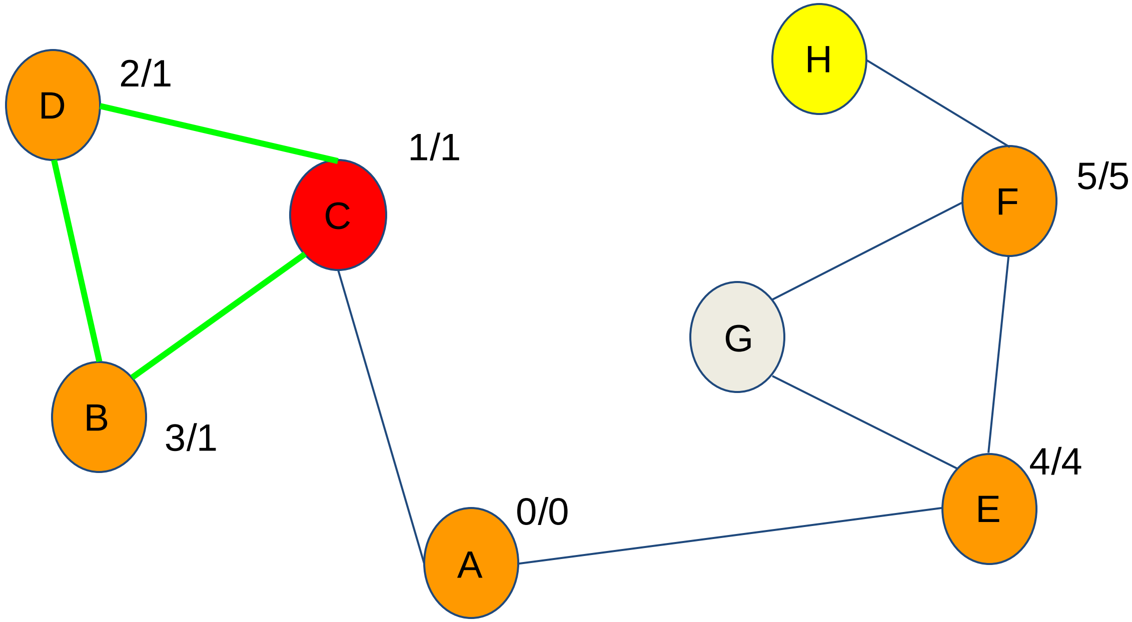
DFS_STACK = (E, 5)
LOWEST = { 0, 1, 1, 1, 4, -1, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, 4, -1, -1, -1 }
VISITED = {1, 1, 1, 1, 1, 0, 0, 0}
PARENTS = { -1, 3, 0, 2, 0, -1, -1, -1 }

Grafos | Puntos de Articulación



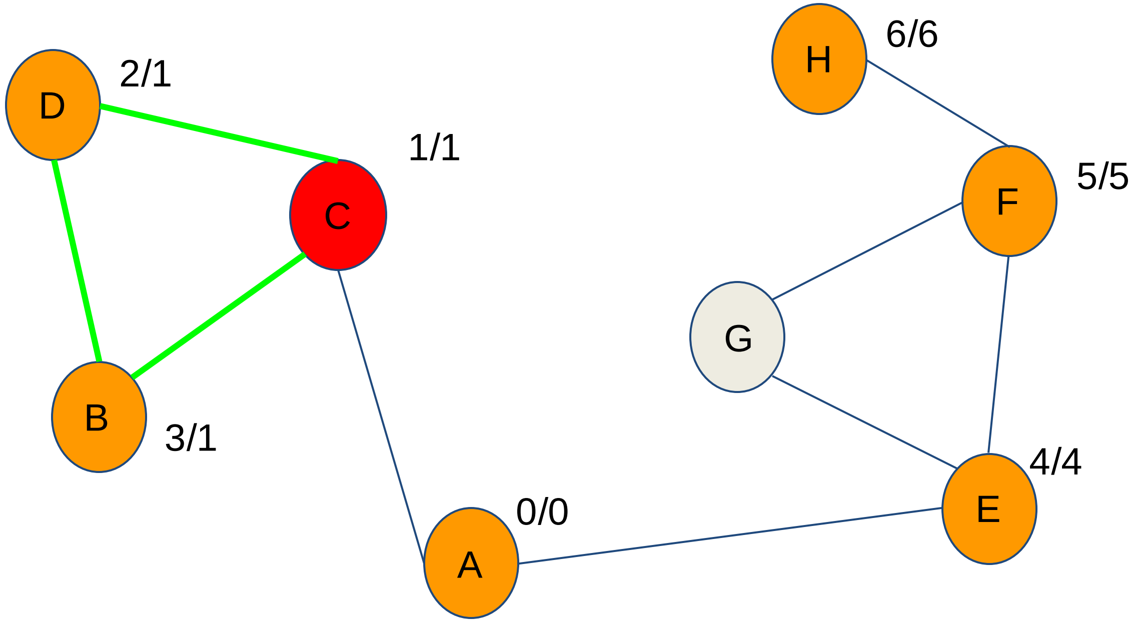
DFS_STACK = (F, 6)
LOWEST = { 0, 1, 1, 1, 4, 5, -1, -1 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, -1 }
VISITED = {1, 1, 1, 1, 1, 1, 0, 0}
PARENTS = { -1, 3, 0, 2, 0, 4, -1, -1 }

Grafos | Puntos de Articulación



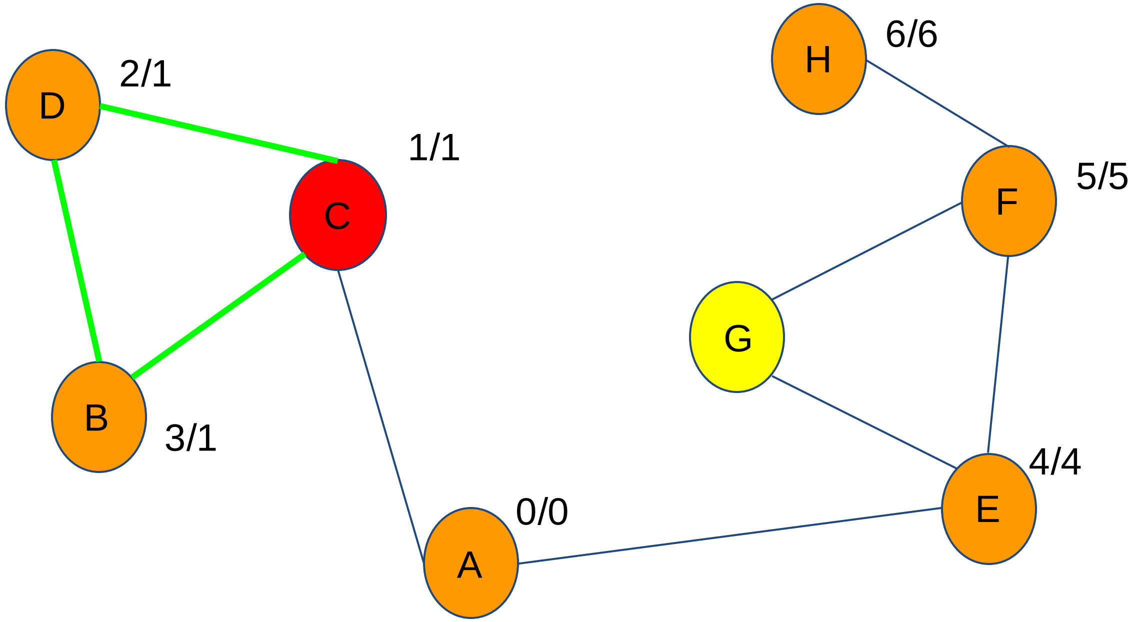
DFS_STACK = (H, 7)
LOWEST = { 0, 1, 1, 1, 4, 5, -1, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 0, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, -1, 5 }

Grafos | Puntos de Articulación



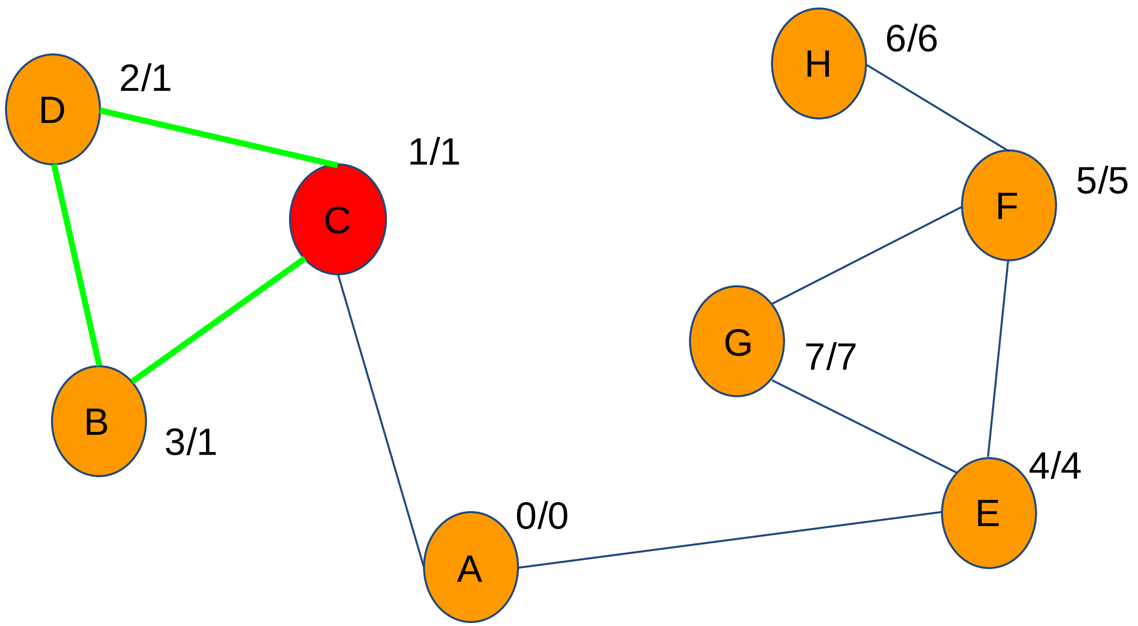
DFS_STACK = (H, 7)
LOWEST = { 0, 1, 1, 1, 4, 5, -1, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 0, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, -1, 5 }

Grafos | Puntos de Articulación



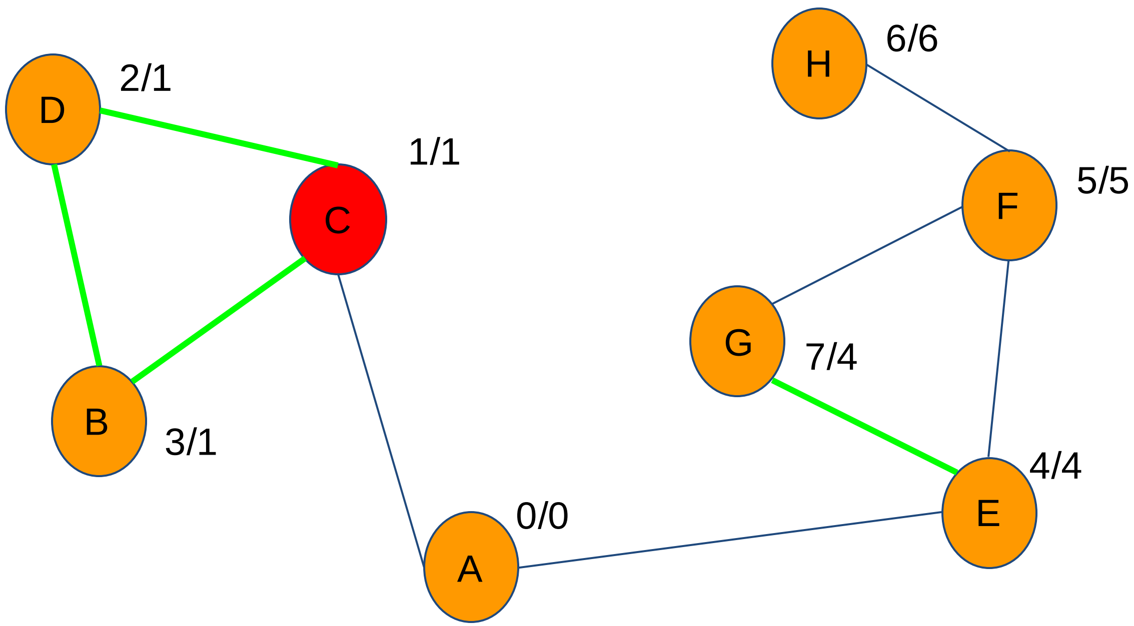
DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 5, 7, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

Grafos | Puntos de Articulación



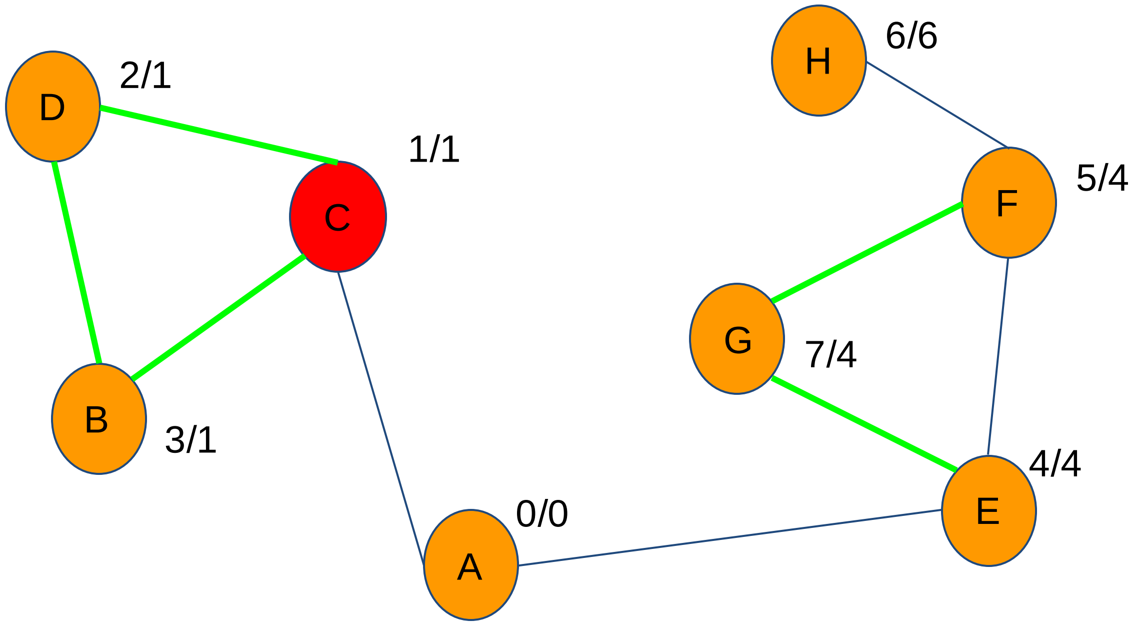
DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 5, 7, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

Grafos | Puntos de Articulación



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 5, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

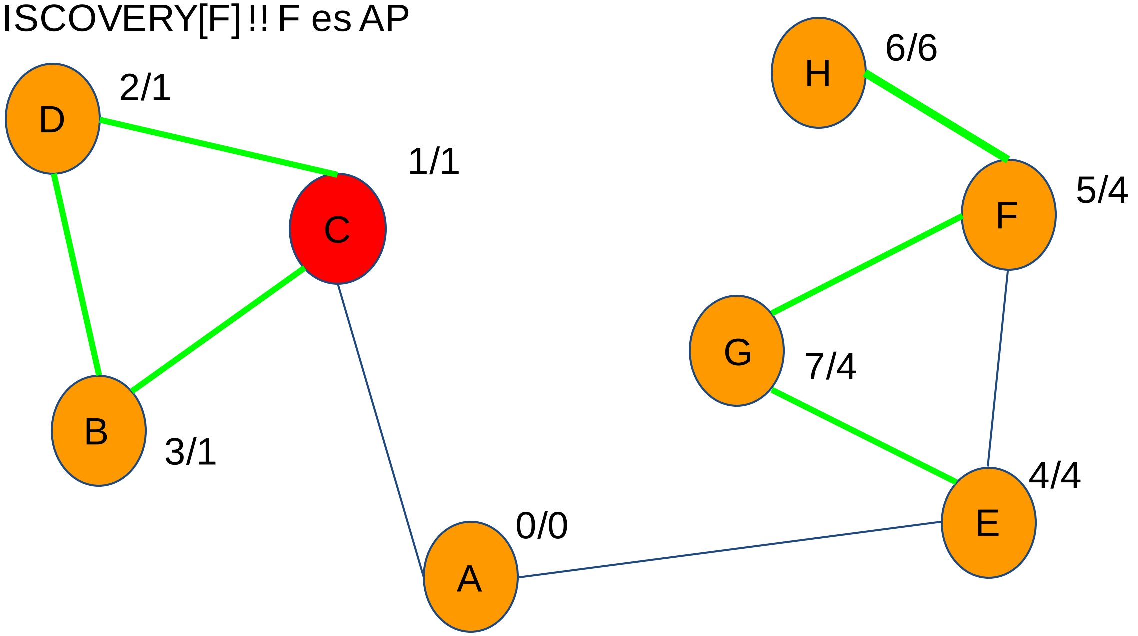
Grafos | Puntos de Articulación



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

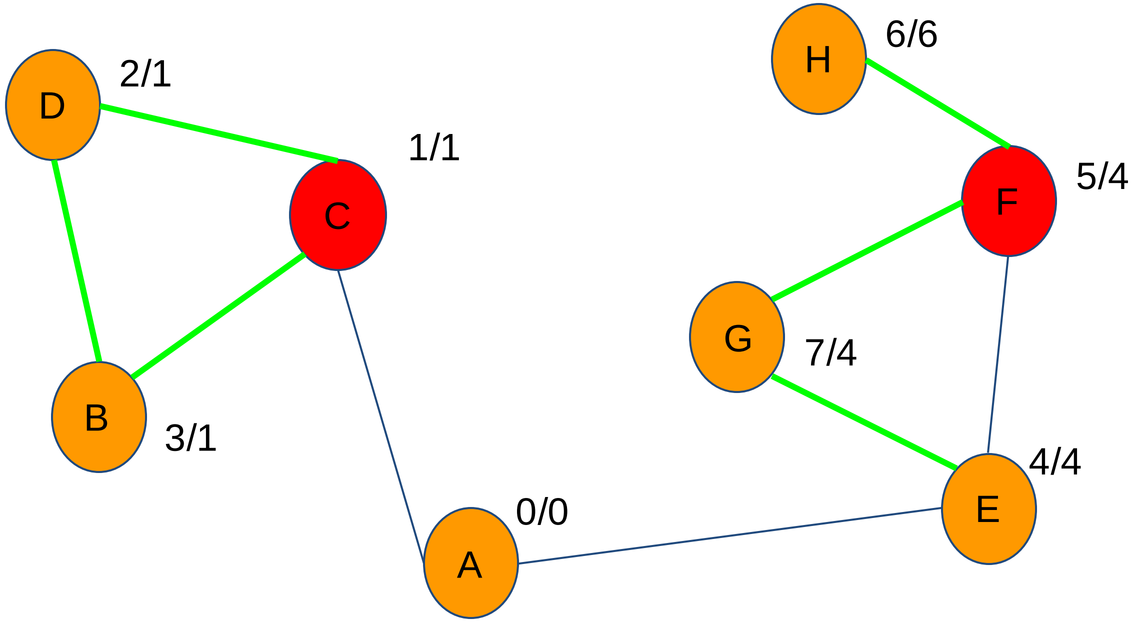
Grafos | Puntos de Articulación

LOWEST[H] >= DISCOVERY[F] !! F es AP



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

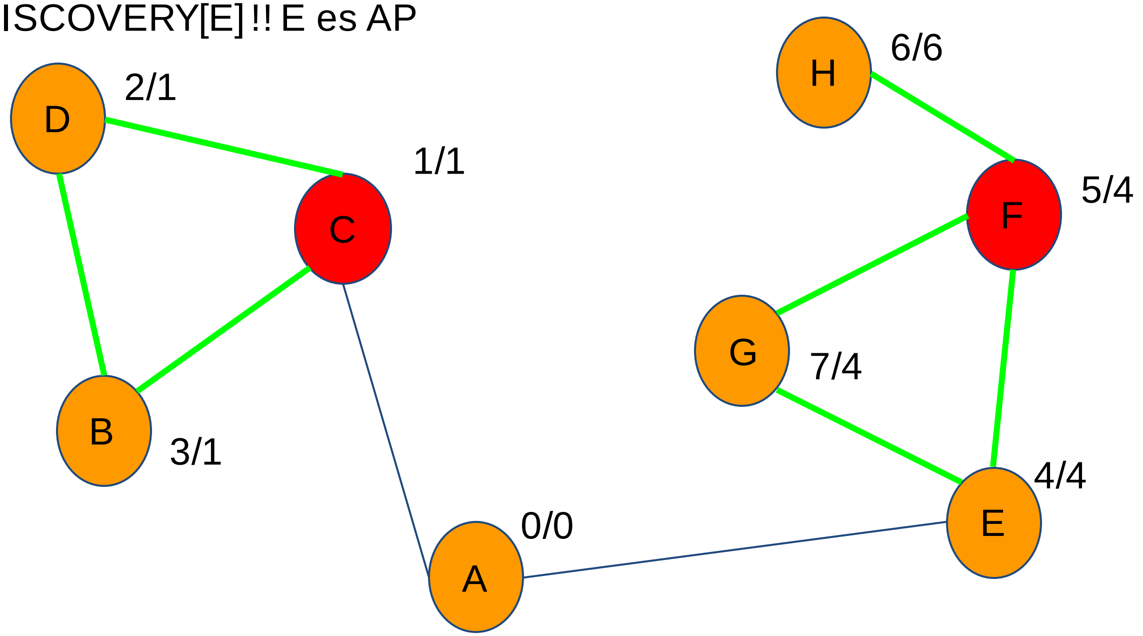
Grafos | Puntos de Articulación



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

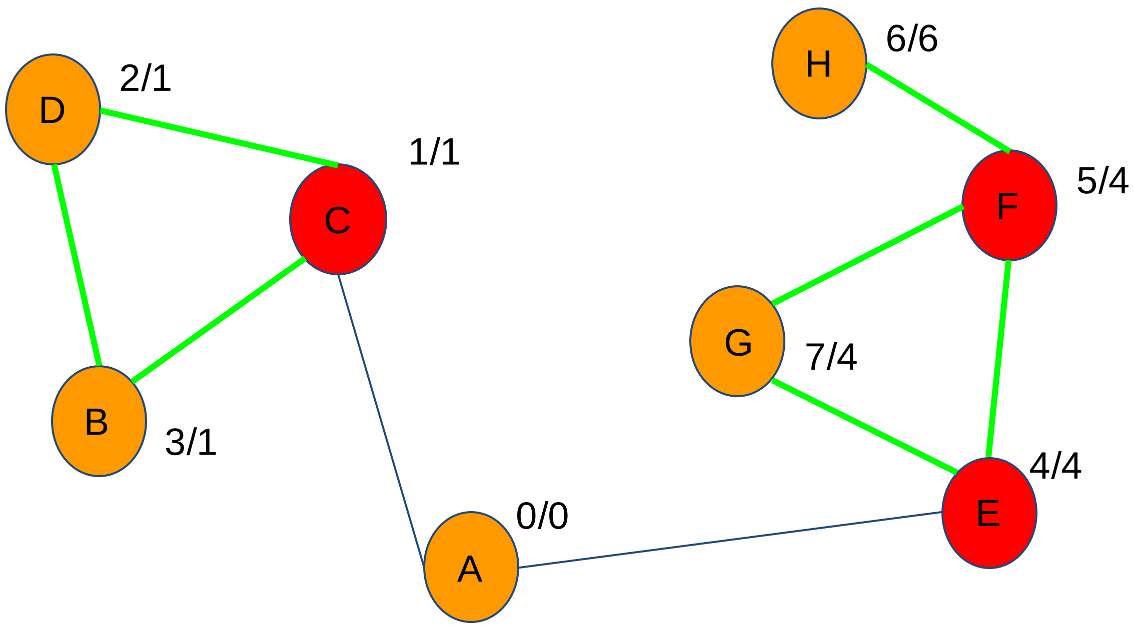
Grafos | Puntos de Articulación

LOWEST[F] >= DISCOVERY[E] !! E es AP



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

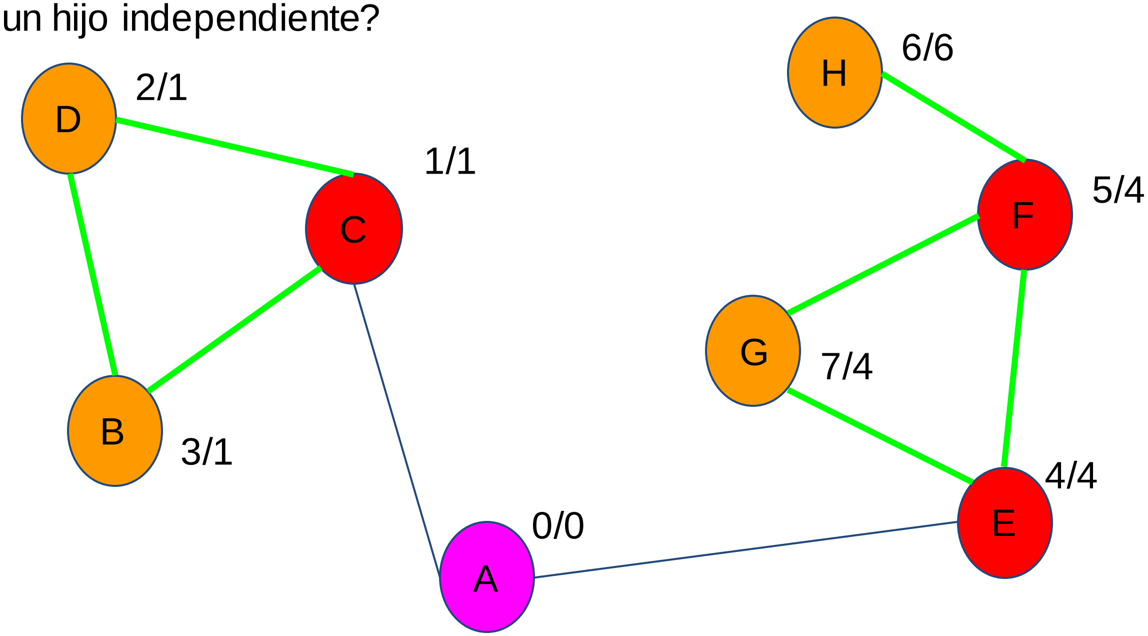
Grafos | Puntos de Articulación



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

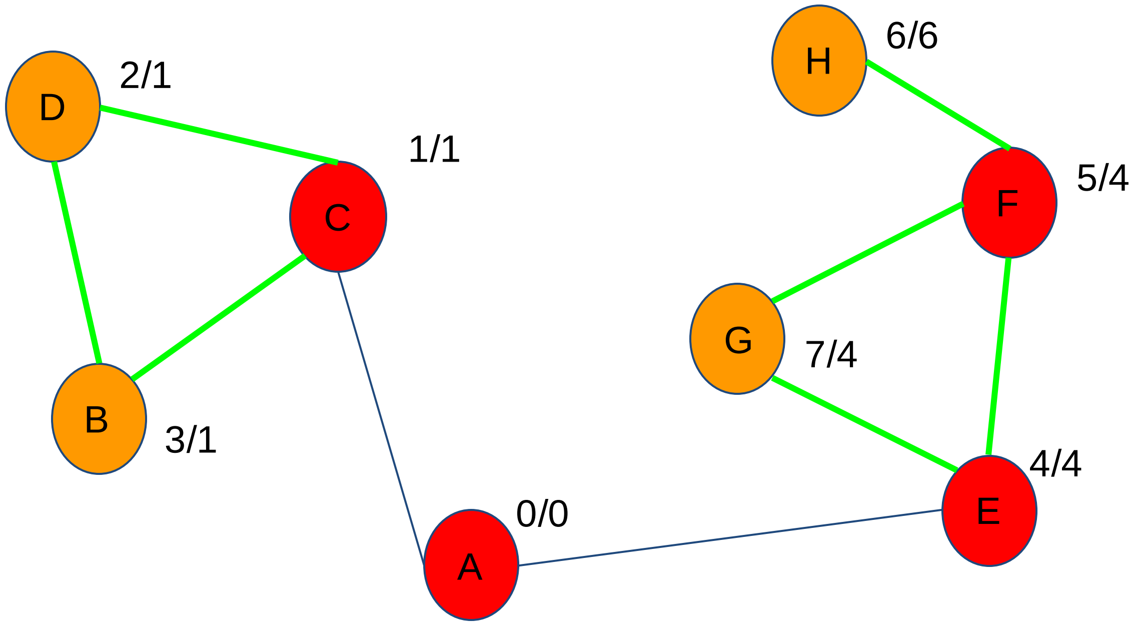
Grafos | Puntos de Articulación

¿A tiene mas de un hijo independiente?



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

Grafos | Puntos de Articulación



DFS_STACK = (G, 8)
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }
VISITED = {1, 1, 1, 1, 1, 1, 1, 1}
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

Grafos | Puntos de Articulación

- Pseudocódigo

```
AP(u, t):
        V(u) = true
        d[u] = low[u] = t++
        children = 0
        for v in edges[u]:
                if(!V(v)):
                        children++, p[v] = u
                        AP(v, t)
                        low[u] = min(low[u], low[v])
                        checkAP(u, v, children)
                else if(v != p[u])
                        low[u] = min(low[u], d[v])
```

Grafos | Puntos de Articulación

- Pseudocódigo

```
checkAP(u, v, children):
        if p[u] == -1 && children > 1:
                ap[u] = true
        if p[u] != -1 && low[v] >= disc[u]:
                ap[u] = true
```

Grafos | Puntos de Articulación

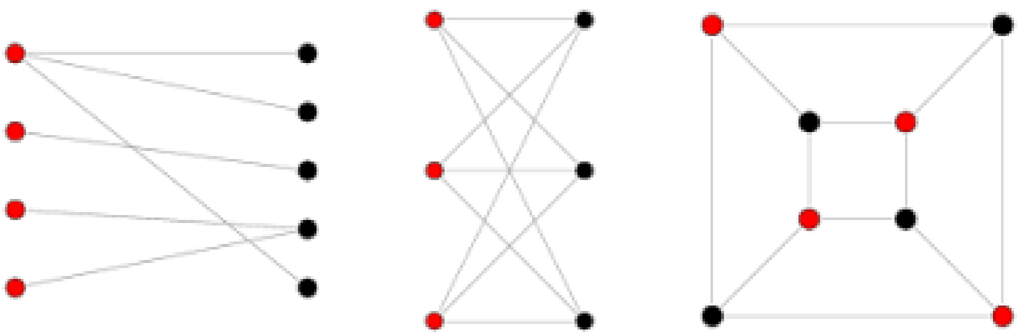
¿Por qué nos interesa comparar si `low[v] >= disc[u]?`

Esta es la condición que nos determina si un nodo es punto de articulación o no. Y ¿**Por qué se cumple**?

Si low[v] fuera menor que disc[u] tendríamos que existe otro nodo w al que v está conectado, de forma que si elimináramos u, no obtendríamos más componentes conexas ya que están conectadas por otra arista.

Grafos - Bipartito

- Un grafo G es bipartito si sus vértices se pueden separar en dos conjuntos disjuntos de manera que las aristas no pueden relacionar vértices de un mismo conjunto

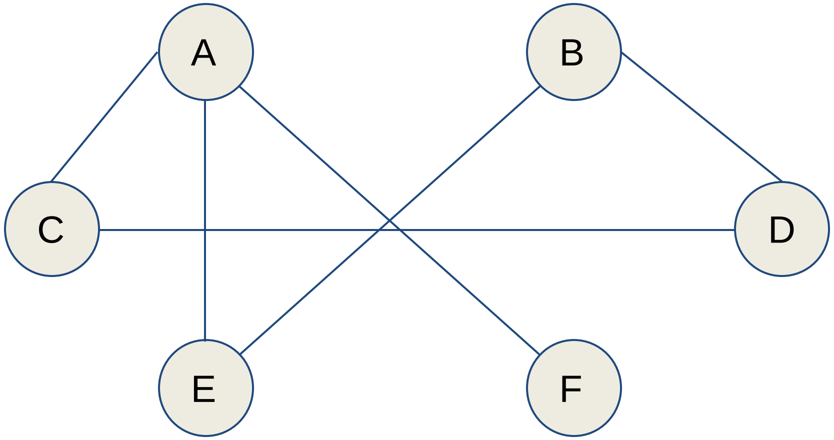


Grafos - Bipartito

- Para probar que un grafo es bipartito solo falta hacer un recorrido dentro del grafo y “pintarlo” con uno de los dos colores, llevando siempre en cuenta que debes cambiar el color por nodo vecino
- Si encuentras un nodo vecino que tenga el mismo color que el actual, ¡**no puede ser un grafo bipartito!**

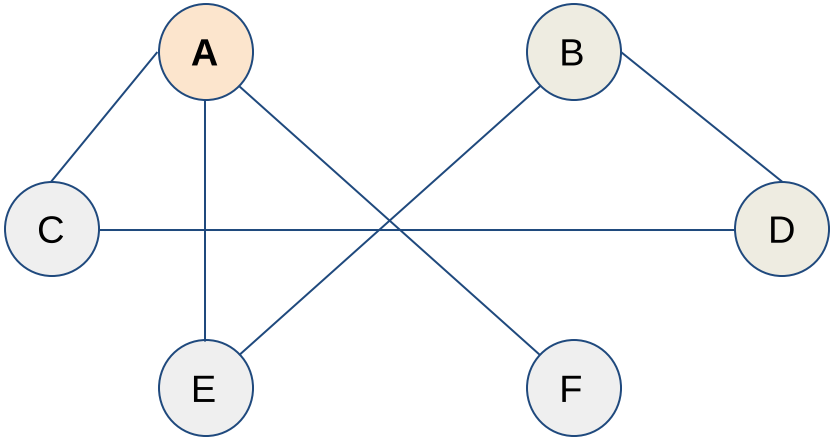
Grafos - Bipartito

Con BFS
iniciando en A
V = [0, 0, 0, 0, 0, 0]
C = [0, 0, 0, 0, 0, 0]
Q = [A]



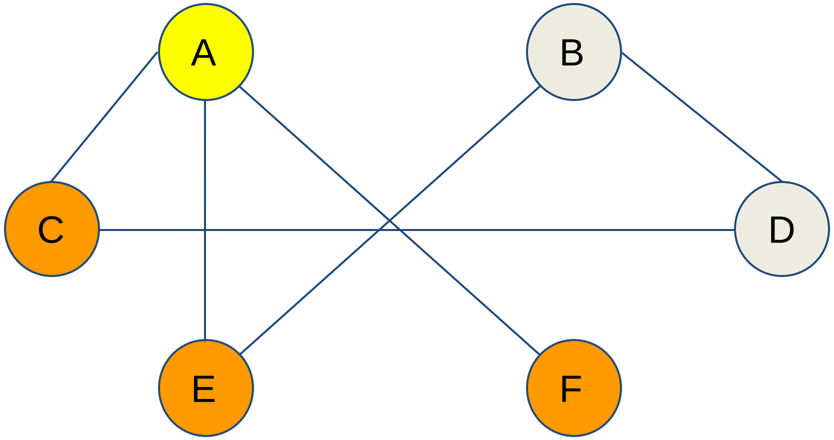
Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 0, 0, 0, 0]
C = [1, 0, 0, 0, 0, 0]
Q = [A]



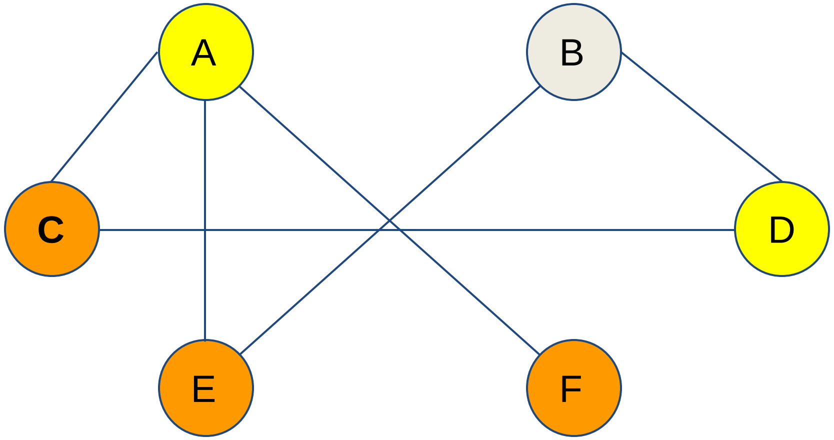
Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 0, 0, 0, 0]
C = [1, 0, 2, 0, 2, 2]
Q = [C, E, F]



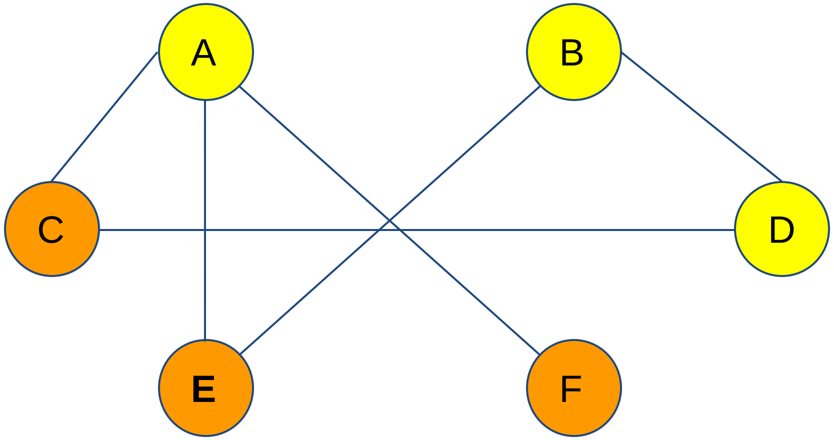
Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 1, 0, 0, 0]
C = [1, 0, 2, 1, 2, 2]
Q = [E, F, D]



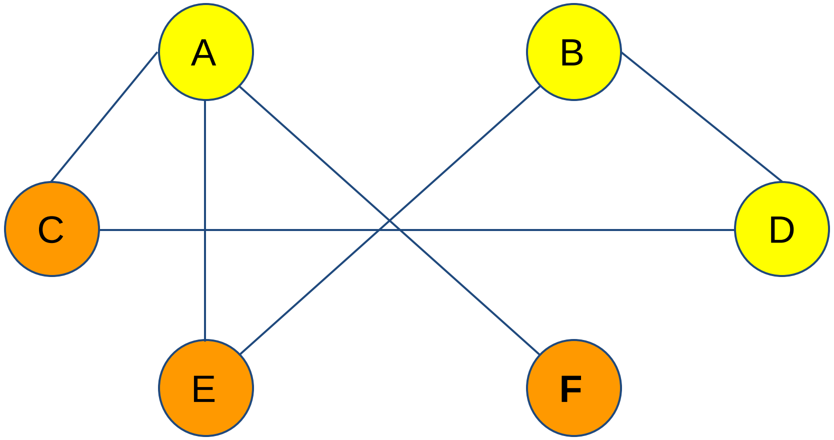
Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 1, 0, 1, 0]
C = [1, 1, 2, 1, 2, 2]
Q = [F, D, B]



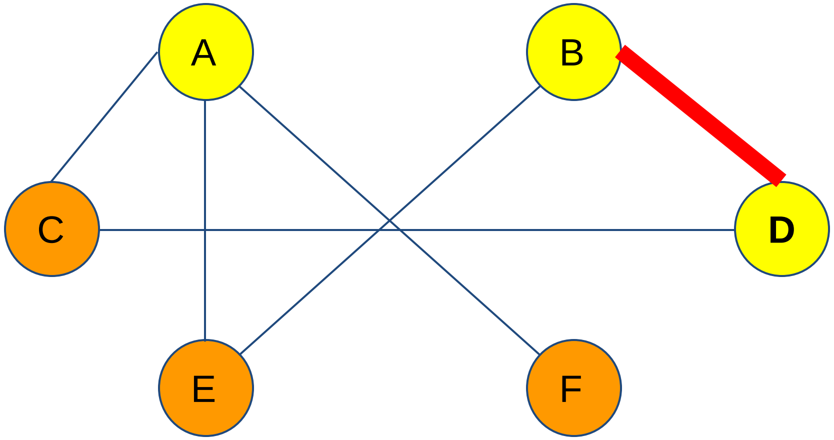
Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 1, 0, 1, 1]
C = [1, 1, 2, 1, 2, 2]
Q = [D, B]



Grafos - Bipartito

Con BFS
iniciando en A
V = [1, 0, 1, 1, 1, 1]
C = [1, 1, 2, 1, 2, 2]
Q = [B]
No es bipartito!



Grafos - Bipartito

- Pseudocódigo

```
isBipartite(init):
  q = [(init, 1)]
  v[init] = true, c[init] = 1
  while !q.empty():
        current = q.top(), q.pop()
        for edge in edges(current):
              neighbor_color = current.c == 1 ? 2 : 1
              if c[edge.dest] == current.c:
                    return false
              if !v[edge.dest]:
                    v[edge.dest] = true
                    c[edge.dest] = neighbor_color
                    q.push((edge.dest, neighbor_color))
  return true
```

Problemas propuestos

BFS / DFS

-AER352

-AER319

-AER537

-Cf Badge

-Cf New Year Transportation

TOPOLOGICAL SORT

Uva10305

PUNTOS DE ARTICULACIÓN

Uva00315

Semana que viene...

- Grafos (parte II)
  - Ponderamiento en grafos
  - Colas de prioridad
  - Algoritmos de distancia mínima (floyd warshall, dijkstra)
  - Estructura Union-Find
  - Árboles de recubrimiento (Prim, Kruskal)

¡Hasta la próxima semana!

Ante cualquier duda sobre el curso o sobre los problemas podéis escribirnos (preferiblemente con copia a algunos / todos los docentes)

- Isaac Lozano (isaac.lozano@urjc.es)
- Raúl Martín (raul.martin@urjc.es)
- Sergio Salazar (s.salazarc.2018@alumnos.urjc.es)
- Francisco Tórtola (f.tortola.2018@alumnos.urjc.es)
- Cristian Pérez (c.perezc.2018@alumnos.urjc.es)
- Xuqiang Liu (x.liu1.2020@alumnos.urjc.es)
- Alicia Pina (a.pinaz.2020@alumnos.urjc.es)
- Sara García (s.garciarod.2020@alumnos.urjc.es)
- **Raúl Fauste** (r.fauste.2020@alumnos.urjc.es)