Introducción a Grafos

Taller de programación competitiva 2025

Miguel Condori — Anahí Sanabria

Universidad Mayor de San Simón

17 de mayo de 2025

¿Qué es un grafo?

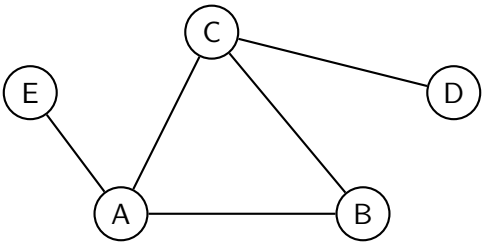
¿Qué es un grafo?

Un grafo es una estructura de datos no lineal. Se representa como:

$$G = (N, A)$$

Donde:
G = grafo
N = conjunto de nodos o vértices
A = conjunto de aristas

Ejemplo de grafo



Vértices o nodos

A, B, C, D y E

Representación de grafos

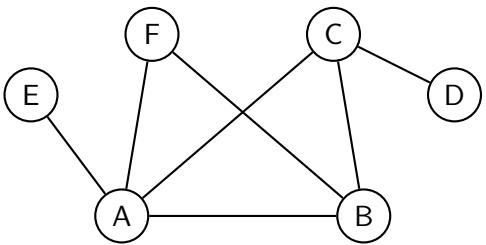
Representación un grafo

No existen los grafos de manera nativa, por lo que se deben implementar :)
Existen dos maneras:

- Listas de adyacencia
- Matriz de adyacencia

Lista de adyacencia

Consiste en armar una lista de todos los vecinos de un nodo i.
Por ejemplo:



La lista sería:
$A \to [E, F, C, B]$
$B \to [A, F, C]$
$C \to [A, B, D]$
$D \to [C]$
$E \to [A]$
$F \to [A, B]$

Entonces:

El *i-ésimo* en la lista contiene otra lista con todos sus nodos vecinos.

Ejemplo en C++

```
vector<vector<int>>gr(n);
for(int i = 0; i<m; i++){
    int a,b; cin>>a>>b;
    a--; b--;
    gr[a].push_back(b);
    gr[b].push_back(a);
}
```

Pasos:

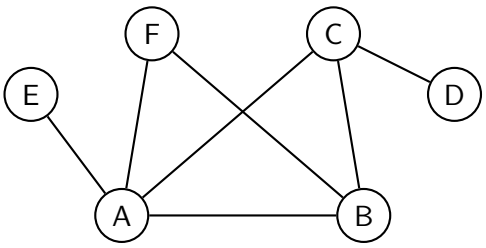
1. Crear el vector de vectores (lista),
2. Leer la cantidad de aristas.
3. Almacenar para la lista de A el nodo B y viceversa.

Notas:

Teniendo a n como número de vértices o nodos y m como cantidad de aristas.

Matriz de adyacencia

Consiste en armar una matriz de n x n tal que cada intersección representa una arista entre nodos. Por ejemplo:



La matriz sería:

	A	B	C	D	E	F
A	0	1	1	0	1	1
B	1	0	1	0	0	1
C	1	1	0	1	0	0
D	0	0	1	0	0	0
E	1	0	0	0	0	0
F	1	1	0	0	0	0

Entonces:

La posición de la matriz $M_{AB} = 1$ representa que hay una arista entre los nodos A y B.

Ejemplo en C++

```
vector<vector<int>>gr(n,vector<int>(n,0));
for(int i = 0; i<m; i++){
    int a,b; cin>>a>>b;
    a--; b--;
    gr[a][b] = 1;
    gr[b][a] = 1;;
}
```

Pasos:

1. Crear la matriz de tamaño $n \times n$,
2. Leer la cantidad de aristas.
3. Almacenar para la lista de A el nodo B y viceversa.

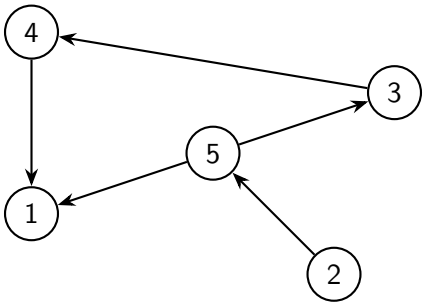
Notas:

Teniendo a n como número de vértices o nodos y m como cantidad de aristas.

Tipos de grafos

Grafo Dirigido

Un grafo dirigido es un grafo que tiene aristas apuntando en una sola dirección. Por ejemplo:

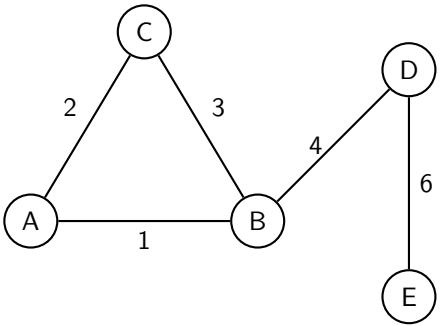


Entonces:

En este tipo de grafos, el nodo 2 va al nodo 5 pero no sucede al revés.

Grafo pesado

Un grafo pesado es aquel que tiene pesos o valores en las aristas. Por ejemplo:

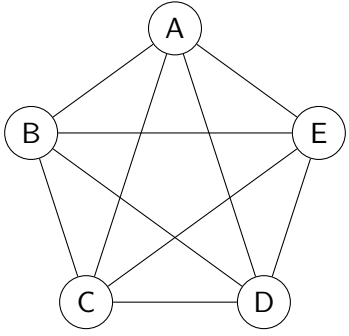


Entonces:

Las aristas pueden representar costos en los grafos. Por ejemplo, un grafo de distancias entre ciudades puede almacenar la distancia de la ciudad A a la ciudad B en sus aristas.

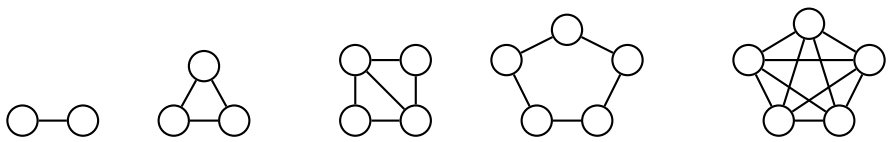
Grafos Completos

- Un **grafo completo** es un grafo en el que **cada par de vértices está conectado por una arista**.
- Se denota como K_n, donde n es el número de vértices.
- Tiene $\frac{n(n-1)}{2}$ aristas en total(n cantidad de vertices).



Ejemplo: K_5

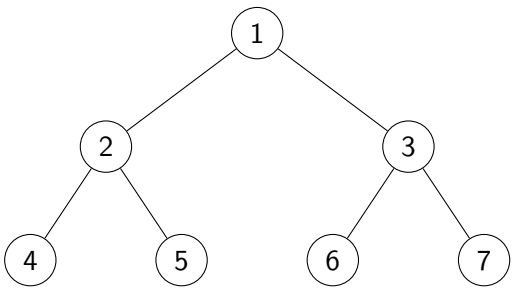
¿Cuál es un grafo completo?



- ¿Cuál o cuáles de estos grafos son completos?
- Un grafo completo tiene todas las conexiones posibles entre sus vértices.

Árboles

Son grafos especiales con propiedades interesantes, por ejemplo: un árbol no puede tener ciclos.



Conceptos clave:
Raíz: Es el nodo que no tiene padre.
Hojas: Son los nodos que no tienen hijos.
Nro. de aristas: siempre es n-1
Grado: el mayor nro de hijos de un nodo.
Amplitud: el nivel más ancho.
Altura: distancia entre el nodo raíz y el nodo hoja más lejano.

Más conceptos sobre grafos

Componentes conexas

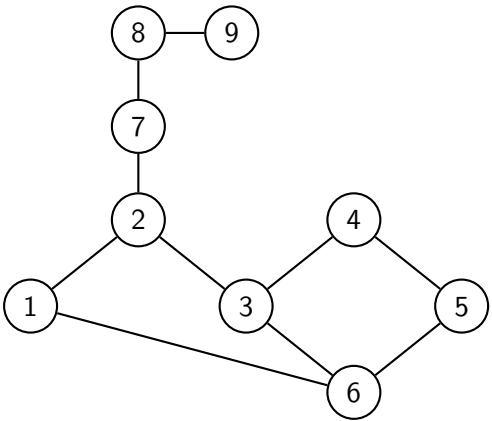
- Un grafo es **conexo** si es posible llegar de cualquier vértice a cualquier otro siguiendo sus aristas.
- Si no es conexo, el grafo está dividido en **componentes conexas**, que son partes del grafo donde sí se cumple esta propiedad.
- Cada componente conexa es un subconjunto de nodos conectados entre sí, pero sin conexión con nodos de otras componentes.
- En un grafo no dirigido, siempre hay al menos una componente conexa (aunque sea un nodo aislado).

Ejemplo: grafo con una sola componente conexa

Entrada:

n = 9, m = 10

- 1 – 2
- 2 – 3
- 3 – 4
- 4 – 5
- 5 – 6
- 6 – 1
- 3 – 6
- 2 – 7
- 7 – 8
- 8 – 9



Componente: {1, 2, 3, 4, 5, 6, 7, 8, 9}

Ejemplo de componentes conexas

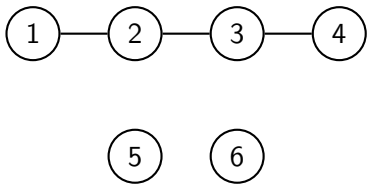
Entrada:

$n = 6$, $m = 3$

- $1 - 2$
- $2 - 3$
- $3 - 4$

Componentes:

- $\{1, 2, 3, 4\}$
- $\{5\}$, $\{6\}$



Ejemplo de componentes conexas (más complejo)

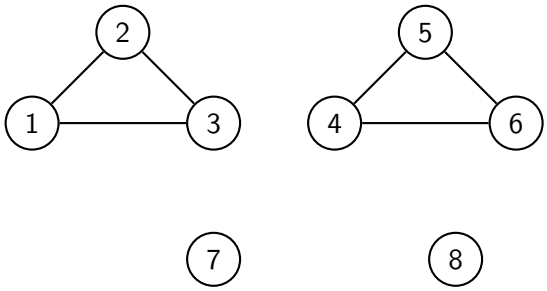
Entrada:

$n = 8$, $m = 6$

- $1 - 2$
- $2 - 3$
- $3 - 1$
- $4 - 5$
- $5 - 6$
- $6 - 4$

Componentes:

- $\{1, 2, 3\}$
- $\{4, 5, 6\}$
- $\{7\}$
- $\{8\}$



Grado de los vértices

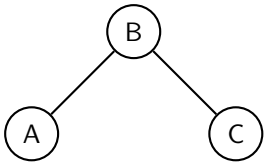
El **grado** de un vértice en un grafo es la cantidad de aristas que inciden sobre él.

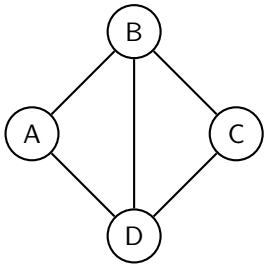
- En un **grafo no dirigido**, el grado de un vértice es el número de vecinos que tiene.
- En un **grafo dirigido**, existen dos tipos de grado:
  - **Grado de entrada (in-degree)**: número de aristas que llegan al vértice.
  - **Grado de salida (out-degree)**: número de aristas que salen del vértice.

Importante:

El grado de un vértice puede usarse para identificar nodos hojas, nodo con mayor grado, nodo con menor grado.

Ejemplos: Grado en grafos no dirigidos

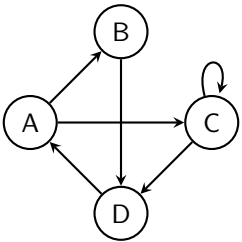




Actividad:

¿Cuál es el grado de cada vértice? ¿Qué pasa con el nodo que tiene un lazo?

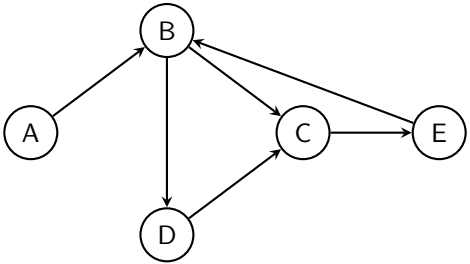
Ejemplos: Grado en grafos dirigidos (in/out)



Actividad:

¿Cuál es el in-degree y out-degree de cada vértice?

Ejemplos: Grado en grafos dirigidos (in/out)



Actividad:

¿Quién tiene más conexiones de salida (out-degree)? ¿Cuál es el vértice con mayor in-degree?

Recorridos en grafos

DFS: Búsqueda por profundidad

El DFS (depth first search) recorre el grafo en profundidad. Utiliza un vector para marcar los nodos visitados.
Complejidad temporal:

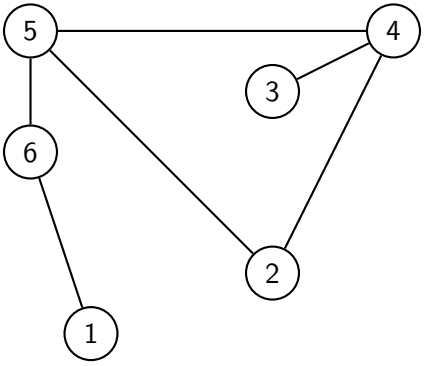
$$O(n + m)$$

Donde n es el número de vértices o nodos y m el número de aristas.

¿Por qué esa complejidad?

El algoritmo es netamente recursivo, por lo que se hace una llamada por cada arista (m) y se procesa una sola vez cada vértice (n). Jamás se vuelve a procesar o marcar un vértice ya procesado.

Ejemplo visual



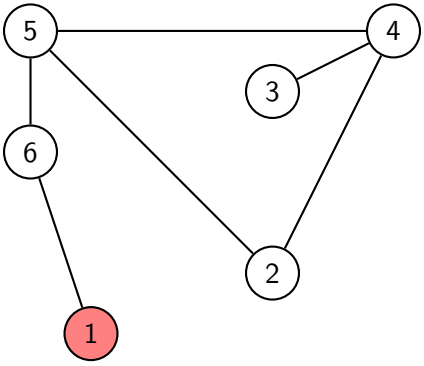
Visitados

[0,0,0,0,0,0]

Recorrido

[]

Ejemplo visual



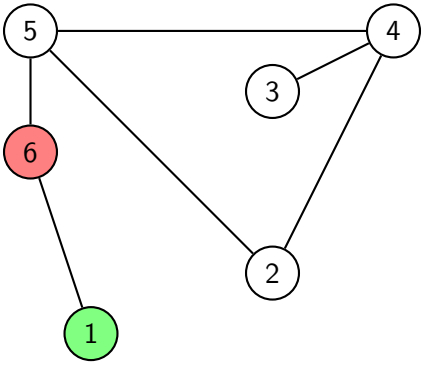
Visitados

[1,0,0,0,0,0]

Recorrido

[1]

Ejemplo visual



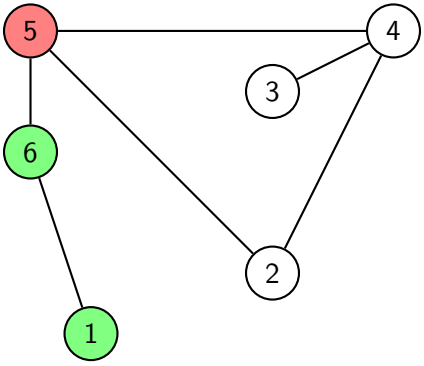
Visitados

[1,0,0,0,0,1]

Recorrido

[1,6]

Ejemplo visual



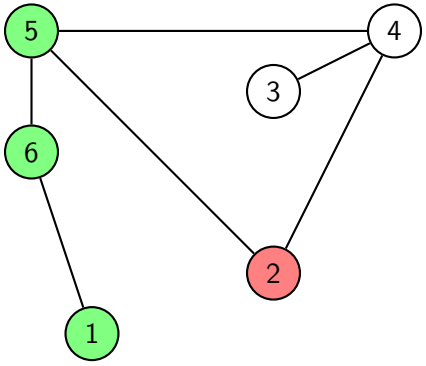
Visitados

[1,0,0,0,1,1]

Recorrido

[1,6,5]

Ejemplo visual



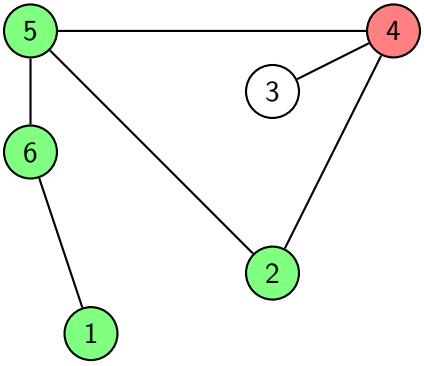
Visitados

[1,1,0,0,1,1]

Recorrido

[1,6,5,2]

Ejemplo visual



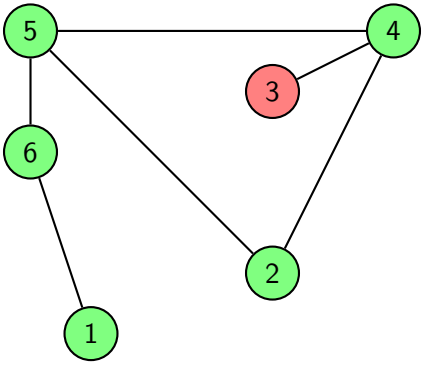
Visitados

[1,0,1,1,1,1]

Recorrido

[1,6,5,2,4]

Ejemplo visual



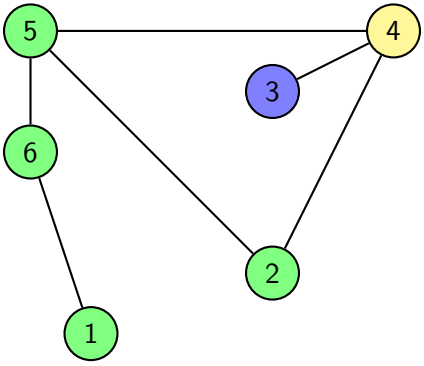
Visitados

[1,1,1,1,1,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



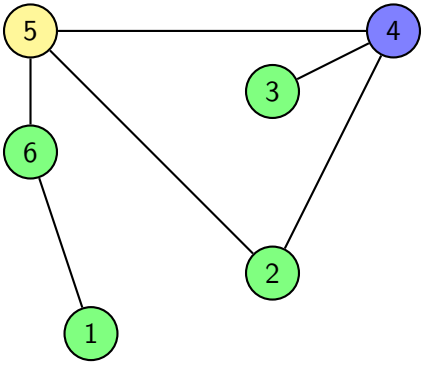
Visitados

[1,1,1,**1**,1,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



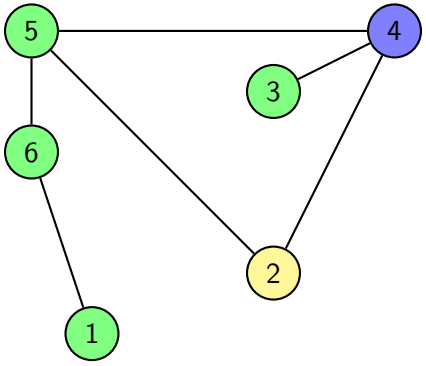
Visitados

[1,1,1,1,**1**,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



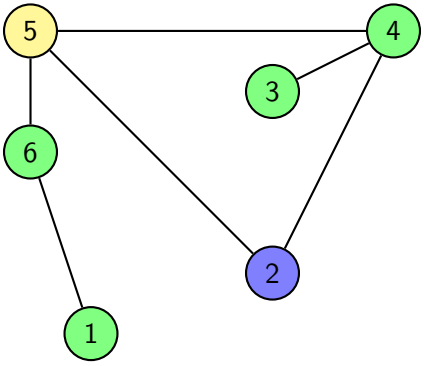
Visitados

[1,**1**,1,1,1,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



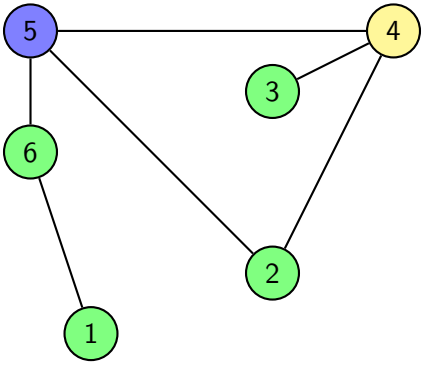
Visitados

[1,1,1,1,**1**,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



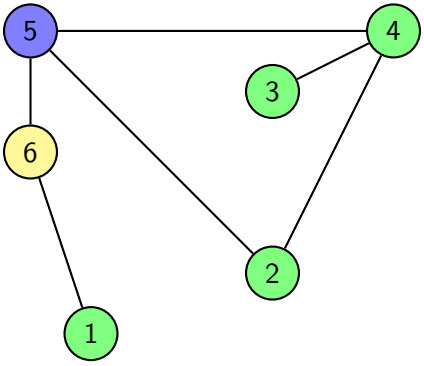
Visitados

[1,1,1,**1**,1,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



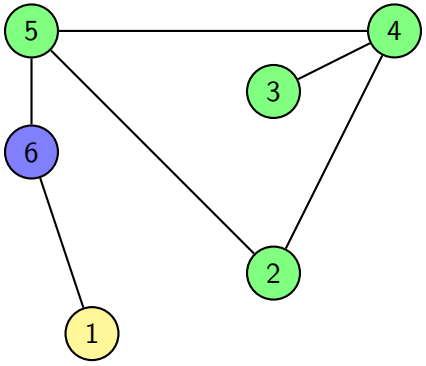
Visitados

[1,1,1,1,1,**1**]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



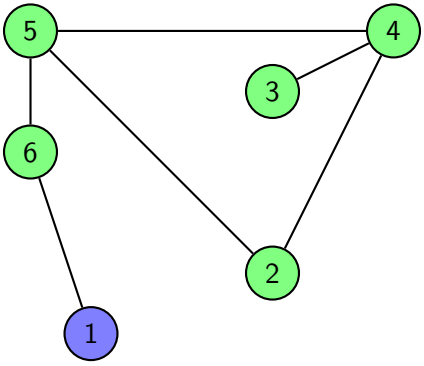
Visitados

[**1**,1,1,1,1,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



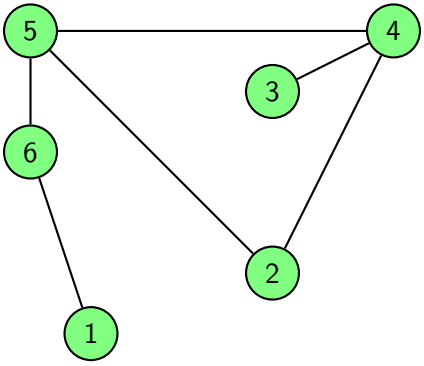
Visitados

[1,1,1,1,**1**,1]

Recorrido

[1,6,5,2,4,3]

Ejemplo visual



Visitados

[1,1,1,1,1,1]

Recorrido

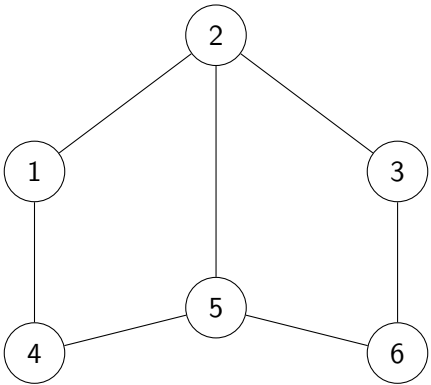
[1,6,5,2,4,3]

Código en C++

```
void dfs(int v){
    visi[v]=true; //primero visitar el nodo
    for(int vi : gr[v]){ //recorrer a sus vecinos
        if(!visi[vi]){ //si el vecino no esta visitado
            dfs(vi); //mandar dfs
        }
    }
}
```

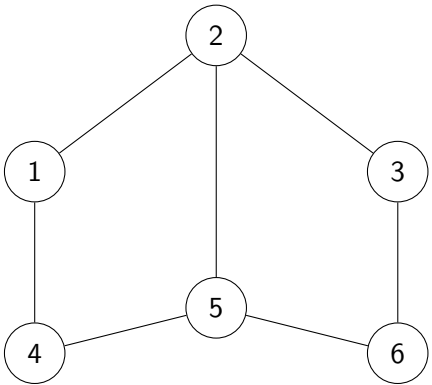
Ejercicio DFS

¿Cuál es el recorrido de este grafo con un dfs desde el nodo 3?



Ejercicio DFS

¿Cuál es el recorrido de este grafo con un dfs desde el nodo 3?



Respuesta:

[3,2,1,4,5,6]
Pueden haber más de una respuesta :)

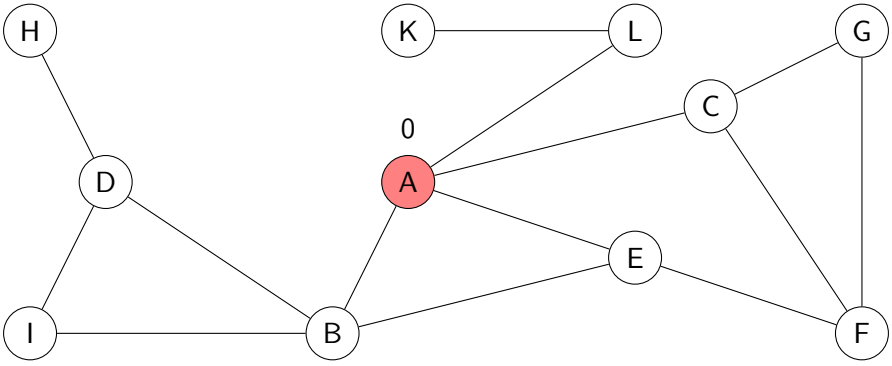
BFS: Búsqueda por Amplitud

- **BFS (Breadth-First Search)** es un algoritmo para recorrer grafos y calcular la **distancia mínima** desde un nodo inicial u hacia los demás nodos.
- Comienza en el nodo u:
  - Se asigna $d[u] = 0$ (distancia mínima a sí mismo).
  - Se marcan como visitados los vecinos directos de u, y se les asigna $d[v] = 1$.
- Luego:
  - Se procesan los nodos con distancia 1.
  - A sus vecinos no visitados se les asigna $d = 2$.
- Este proceso continúa por niveles: para encontrar los nodos a distancia $k + 1$, se exploran los vecinos de los nodos a distancia k que aún no han sido visitados.
- BFS garantiza encontrar el camino más corto en cantidad de aristas en grafos no pesados.

Pasos del algoritmo BFS (más claro)

- Paso 1: Elegimos un nodo inicial u, lo encolamos y le asignamos distancia $d[u] = 0$.
- Paso 2: Mientras la cola no esté vacía, sacamos el primer nodo de la cola (llamémoslo u).
- Paso 3: Miramos a cada vecino v_i de u. Si v_i no fue visitado:
  - Lo encolamos
  - Le asignamos distancia $d[v_i] = d[u] + 1$
- Paso 4: Repetimos esto hasta que la cola esté vacía.
- Paso 5: Al final, cada $d[v_i]$ nos dice la distancia más corta desde el nodo u hasta ese nodo.

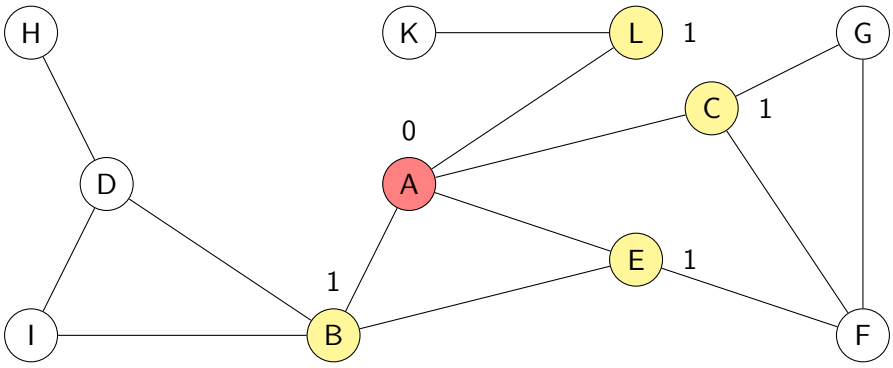
Ejemplo de grafo



Cola de Vértices

{A}

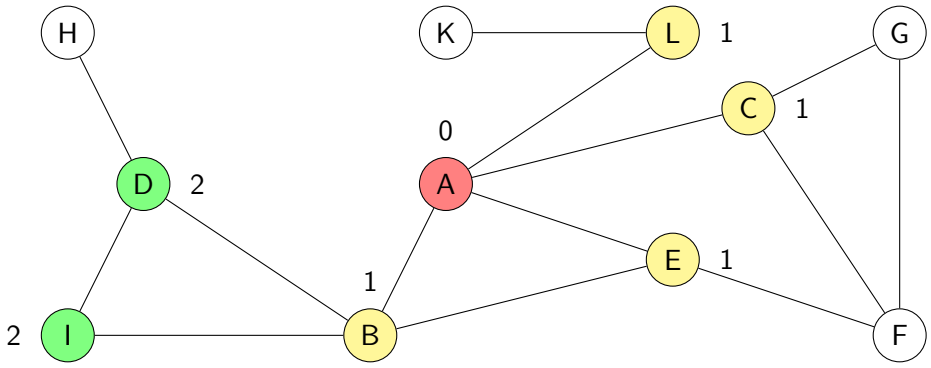
Ejemplo de grafo



Cola de Vértices

{B,E,C,L} Sacamos el vértice {A}

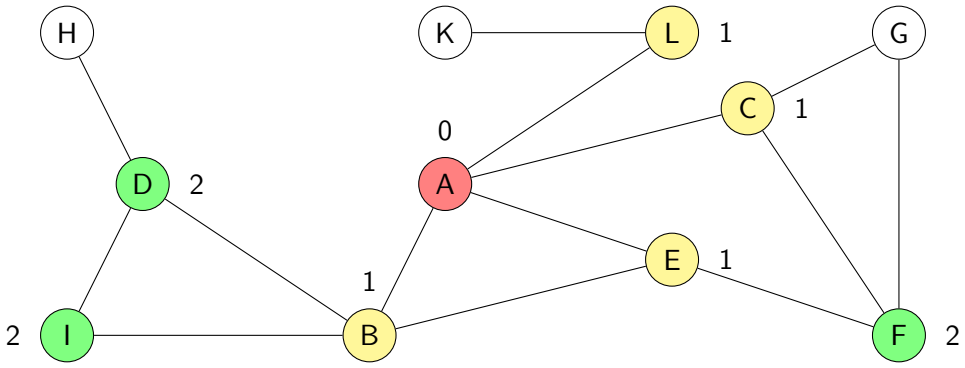
Ejemplo de grafo



Cola de Vértices

{E,C,L,I,D} Sacamos el vértice {B}

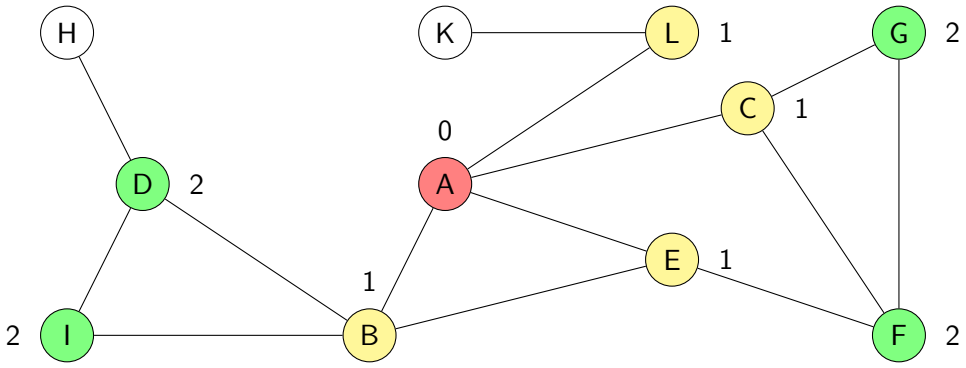
Ejemplo de grafo



Cola de Vértices

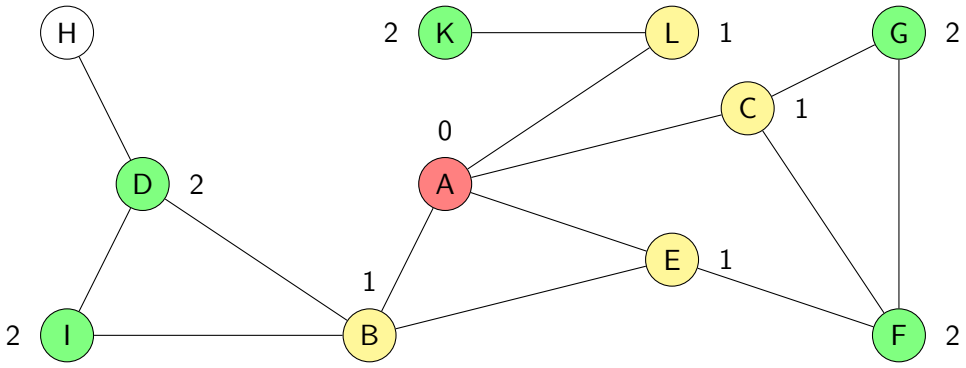
{C,L,I,D,F} Sacamos el vértice {E}

Ejemplo de grafo



Cola de Vértices	
{L,I,D,F,G}	Sacamos el vértice {C}

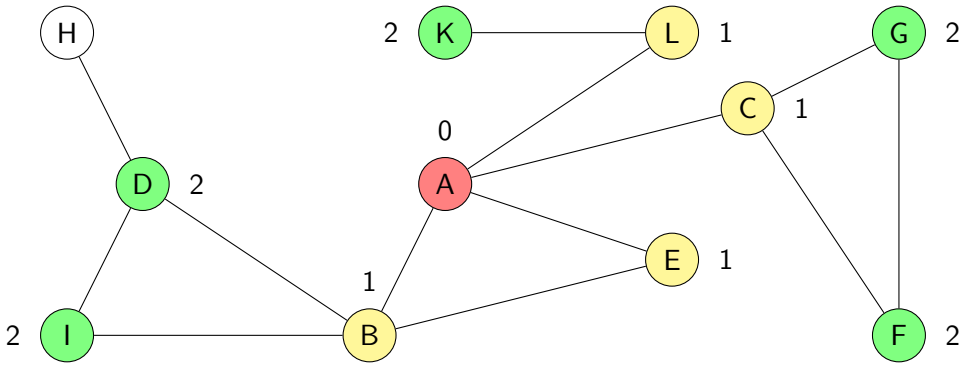
Ejemplo de grafo



Cola de Vértices

{I,D,F,G,K} Sacamos el vértice {L}

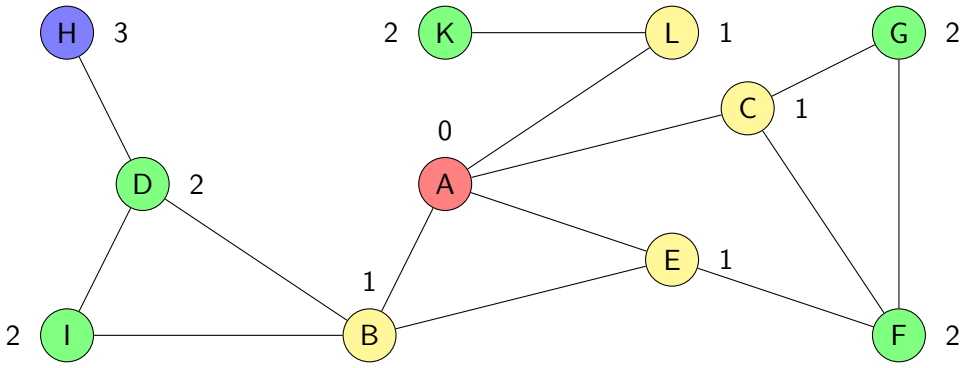
Ejemplo de grafo



Cola de Vértices

{D,F,G,K} Sacamos el vértice {I}

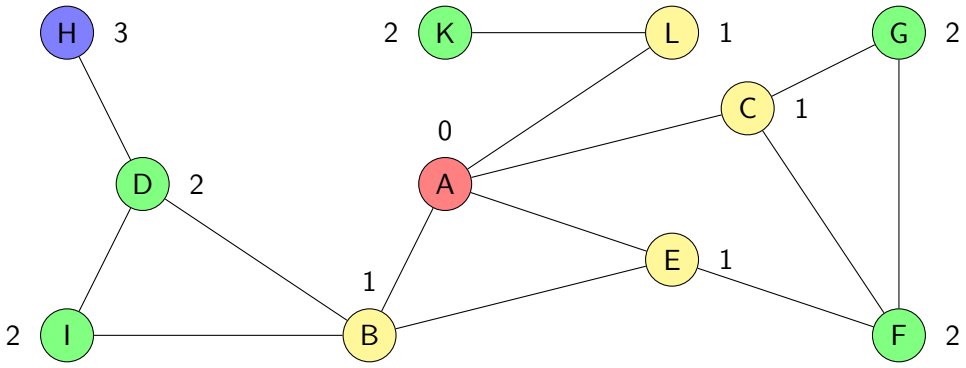
Ejemplo de grafo



Cola de Vértices

{F,G,K,H} Sacamos el vértice {D}

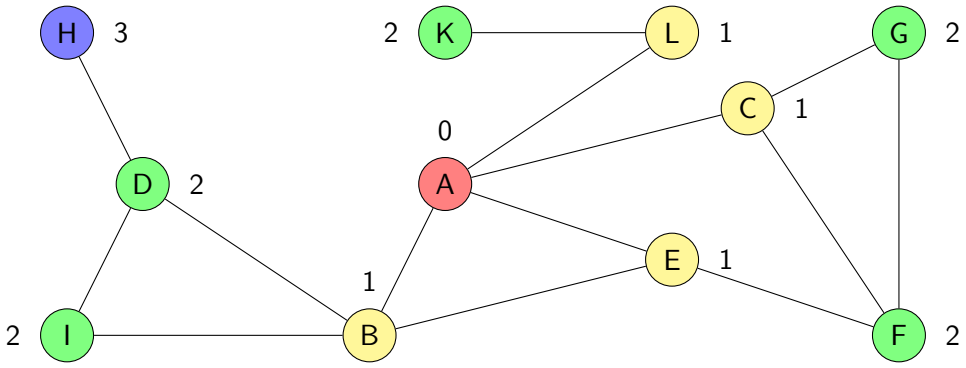
Ejemplo de grafo



Cola de Vértices

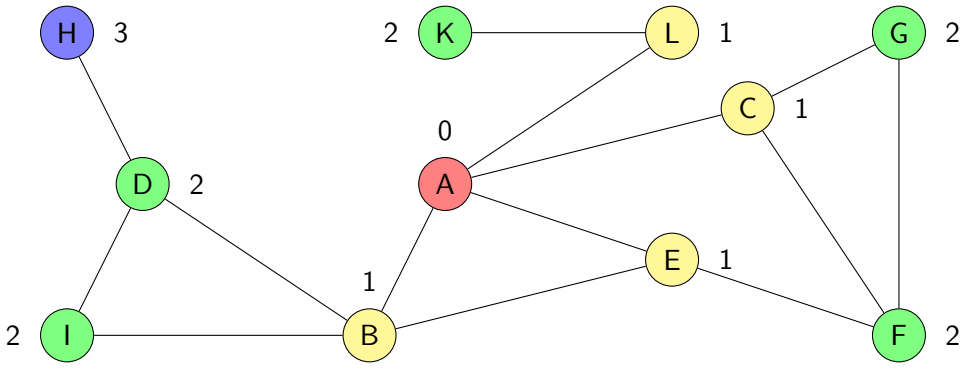
{G,K,H} Sacamos el vértice {F}

Ejemplo de grafo



Cola de Vértices	
{K,H}	Sacamos el vértice {G}

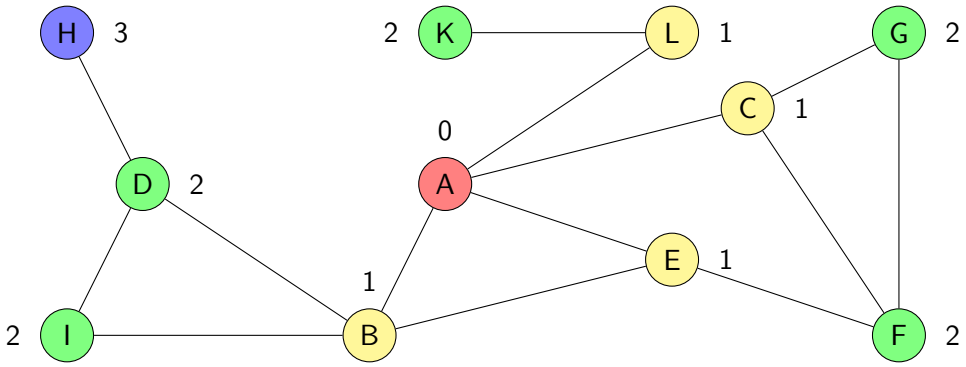
Ejemplo de grafo



Cola de Vértices

{H}	Sacamos el vértice {K}

Ejemplo de grafo



Cola de Vértices

{} Sacamos el vértice {H}

Código en C++

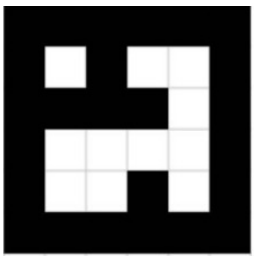
```
vector<int> bfs(vector<vector<int>>& g , int u) {
    vector<int> dis(g.size(), -1); // vector de distancias
    queue<int> q;
    dis[u] = 0;
    q.push(u);
    while (!q.empty()) { // mientras haya nodos por procesar
        int node = q.front();
        q.pop();
        for (int v : g[node]) { // para cada v vecino de node
            if (dis[v] == -1) { // si v no fue encolado aun
                dis[v] = dis[node] + 1;
                q.push(v);
            }
        }
    }
    return dis;
}
```

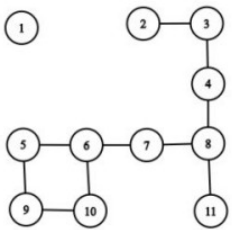
Complejidad del algoritmo BFS

- **Complejidad temporal:** $\mathcal{O}(V + E)$, donde:
  - V es el número de vértices (nodos) del grafo.
  - E es el número de aristas (conexiones) del grafo.
- **¿Por qué es $\mathcal{O}(V + E)$?**
  - Cada vértice se encola y se procesa una sola vez.
  - Al procesar un vértice, se recorren todas sus aristas adyacentes.
- **Operaciones de cola:**
  - Las operaciones de encolar y desencolar tienen complejidad $\mathcal{O}(1)$.
- **Conclusión:** El tiempo total depende linealmente del número de nodos y aristas, lo que hace que BFS sea eficiente incluso en grafos grandes.

Flood Fill

Dado un tablero donde un 0 representa el color negro y un 1 el color blanco.
Contar la cantidad de componentes blancas, y cuantas celdas tiene cada una.





Se puede usar la matriz como grafo implícito

Flood Fill en C++

```
int n, m; // dimensiones del tablero
int dir[2][4] = {{0, 0, 1, -1},
                 {1, -1, 0, 0}};
// direcciones: derecha,izquierda,abajo,arriba
vector<vector<int>> tab, visi;

int floodfill(int x, int y) {
    if (x < 0 || y < 0 || x >= n || y >= m || visi[x][y] ||tab[x][y] == 0)
        return 0;
    visi[x][y] = 1;
    int ret = 1;
    for (int i = 0; i < 4; i++)
        ret += floodfill(x + dir[0][i], y + dir[1][i]);
    return ret;
}
```

Ejercicios de práctica

Dfs
https://cses.fi/problemset/task/1666

Bfs
https://cses.fi/problemset/task/1667/

Flood fill
https://cses.fi/problemset/task/1192

Gracias

Codeforces: Simurdiera_MAC , AnahiSU