Grafos I

Nicolás Rojas Bustos

nicolas.rojas11@alumnos.ucn.cl

Bloque 1 · Fundamentos

1 · Fundamentos

2 · Recorridos

Ruta de la clase

Parte 1 — la base

- Qué es un grafo y cómo se ve en ProgComp
- Matriz vs lista de adyacencia (y sus complejidades)
- Grado, subgrafos, recorridos, conexidad y componentes
- Árboles y bosques

Parte 2 — recorridos

- Digrafos
- DFS y BFS
- Implementacion
- Distancias

¿Grafos?

- Los grafos son un **modelo matemático** ampliamente utilizado.
- Permiten modelar **conexiones e interacciones** entre distintos entes de un sistema.

Redes sociales, mapas de ciudades, dependencias entre tareas, moléculas, la malla de la U... casi todo se puede pensar como un grafo.

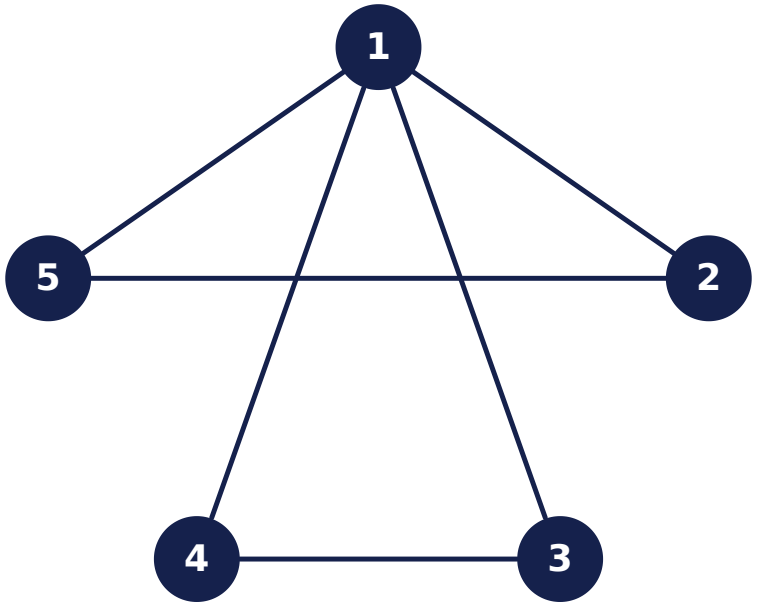
Definición

Un **grafo** es un par ordenado $G = (V, E)$, donde:

- V es un conjunto finito y no vacío, cuyos elementos llamamos **vértices**.
- $E \subseteq V \times V$, cuyos elementos llamamos **aristas**.

Una arista (u, v) conecta dos vértices. Si no importa el orden ((u, v) es lo mismo que (v, u)), el grafo es **no dirigido**.

Ejemplo



$$V = \{1, 2, 3, 4, 5\}, \quad E = \{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (5, 2)\}$$

¿Cómo leo un grafo en ProgComp?

Formato típico: primero $n = |V|$ y $m = |E|$, luego m líneas con las aristas.

```
int n, m;
cin >> n >> m;
for (int e = 0; e < m; ++e) {
    int u, v;           // E_e = (u, v)
    cin >> u >> v;
    // ... aquí la guardamos (ya veremos cómo)
}
```

Los vértices suelen venir en [1..n]. En C++ es cómodo pasarlos a [0..n-1] con `u--; v--;`

¿Y cómo los almaceno?

Dos formas clásicas:

- **Matriz de adyacencia** — una tabla $n \times n$.
- **Lista de adyacencia** — para cada vértice, la lista de sus vecinos.

Cada una tiene ventajas y desventajas. Las vemos y después comparamos.

Matriz de adyacencia

`mtz[u][v] = 1` si existe la arista (u, v), y `0` si no. Como el grafo es no dirigido, la matriz es **simétrica**.

	1	**2**	**3**	**4**	**5**
1	0	1	1	1	1
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	0
5	1	1	0	0	0

¿Cómo creo una matriz de adyacencia?

```
int n, m;
cin >> n >> m;
vector<vector<int>> mtz_ady(n, vector<int>(n, 0));
for (int e = 0; e < m; ++e) {
    int u, v;
    cin >> u >> v;
    u--; v--;                    // pasamos a [0..n-1]
    // E es un conjunto de pares NO ordenados,
    // agregamos la arista en ambos sentidos
    mtz_ady[u][v] = 1;
    mtz_ady[v][u] = 1;
}
```

Lista de adyacencia

Para cada vértice guardamos **solo** sus vecinos. Mucho más liviana cuando el grafo tiene pocas aristas (que es lo común).

```
1 → 2, 3, 4, 5
2 → 1, 5
3 → 1, 4
4 → 1, 3
5 → 1, 2
```

¿Cómo creo una lista de adyacencia?

```cpp
int n, m;
cin >> n >> m;
vector<vector<int>> lst_ady(n);
for (int e = 0; e < m; ++e) {
    int u, v;
    cin >> u >> v;
    u--; v--;
    // agregamos la arista en ambos sentidos
    lst_ady[u].push_back(v);
    lst_ady[v].push_back(u);
}
```

Matriz vs Lista: ¿cuál conviene?

Operación	Matriz	Lista
Crear la estructura	$O(V^2)$	$O(V+E)$
¿Existe la arista (u, v)?	$O(1)$	$O(\text{grado}(u))$
¿Cuántos vecinos tiene v?	$O(V)$	$O(1)$
Recorrer los vecinos de v	$O(V)$	$O(\text{grado}(v))$
Memoria	$O(V^2)$	$O(V+E)$

Regla de bolsillo: en ProgComp casi siempre se usa **lista**. La matriz sirve cuando el grafo es chico ($V \lesssim 500$) o cuando preguntamos mucho "¿existe la arista?".

Grado de un vértice

El **grado** de v, $\deg(v)$, es la cantidad de aristas que tocan a v.

- En una lista de adyacencia: `lst_ady[v].size()`.
- **Lema del apretón de manos:** $\sum_{v \in V} \deg(v) = 2\,|E|$.

En digrafos distinguimos **grado de entrada** (in-degree) y **grado de salida** (out-degree).

Ejemplo en vivo 1

Leamos y representemos este grafo (matriz **y** lista):

```
10 8
1 2      3 4      1 5      7 8
3 2      6 8      5 4      6 7
```

¿Cuántos vértices quedan "solos"? ¿Cómo se ve la lista de adyacencia?

Subgrafos

Un **subgrafo** $H = (V', E')$ de $G = (V, E)$ cumple:

- $V' \subseteq V$
- $E' \subseteq E$ (y cada arista de E' conecta vértices que están en V')

Dicho simple: es "un pedazo" del grafo original.

Recorridos, caminos y ciclos

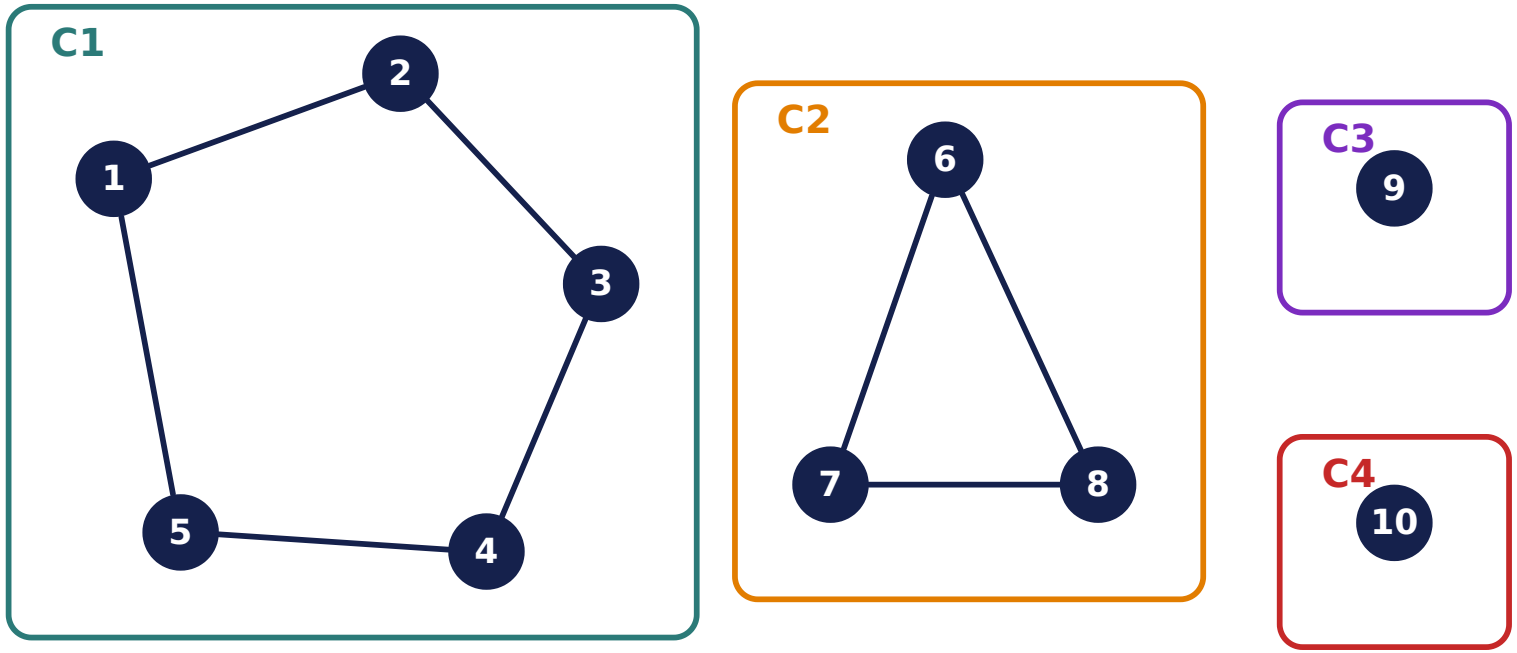
Un **recorrido** de u a v es una secuencia $(v_0, v_1, \ldots, v_k)$ con $v_0 = u$, $v_k = v$, donde cada par consecutivo es una arista.

- **Circuito:** recorrido con $v_0 = v_k$ (empieza y termina en el mismo vértice).
- **Camino:** recorrido **sin vértices repetidos**.
- **Ciclo:** circuito sin vértices repetidos (salvo $v_0 = v_k$).

Conexidad

- Un grafo es **conexo** si es trivial o si $\forall\, u, v \in V$ existe un camino entre ellos.
- Si no lo es, se dice **disconexo**.
- Una **componente conexa** es un subgrafo conexo **maximal**: no existe otro subgrafo conexo que lo contenga estrictamente.

Componentes conexas — ejemplo en vivo 1



Resumiendo el ejemplo

- G es **disconexo**: tiene **4 componentes conexas**.
- $C_1 = \{1, 2, 3, 4, 5\}$ es un **ciclo** C_5 (cada vértice tiene 2 vecinos $\rightarrow$ *2-regular*).
- $C_2 = \{6, 7, 8\}$ es el grafo **completo** K_3 (todos con todos).
- $C_3 = \{9\}$ y $C_4 = \{10\}$ son grafos **triviales** (un vértice, sin aristas).

Contar componentes conexas

Un DFS/BFS visita **toda** una componente. Basta lanzar uno nuevo por cada vértice sin visitar:

```
int componentes = 0;
for (int u = 0; u < n; ++u) {
    if (!visitado[u]) {
        componentes++;
        dfs(u);           // marca toda la componente de u
    }
}
```

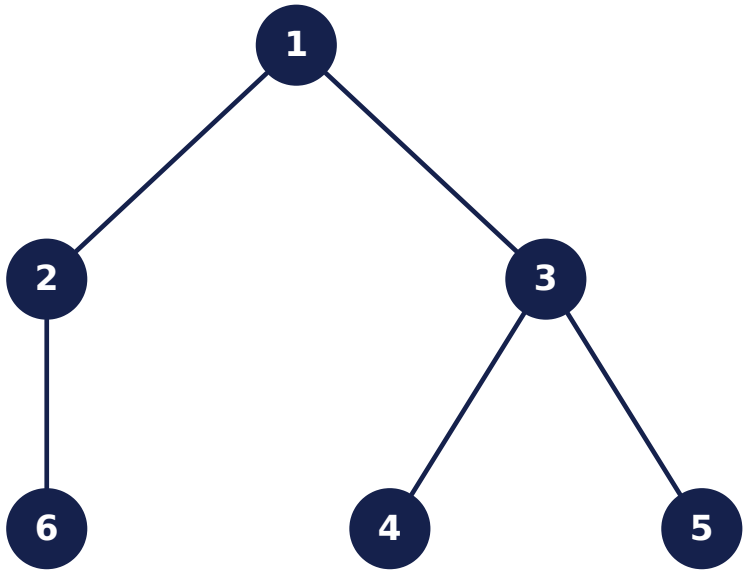
Este patrón "por cada vértice sin visitar, lanzo un recorrido" aparece en varios problemas.

Árboles

Si G es **conexo y sin ciclos**, decimos que G es un **árbol**. Las siguientes proposiciones son **equivalentes**:

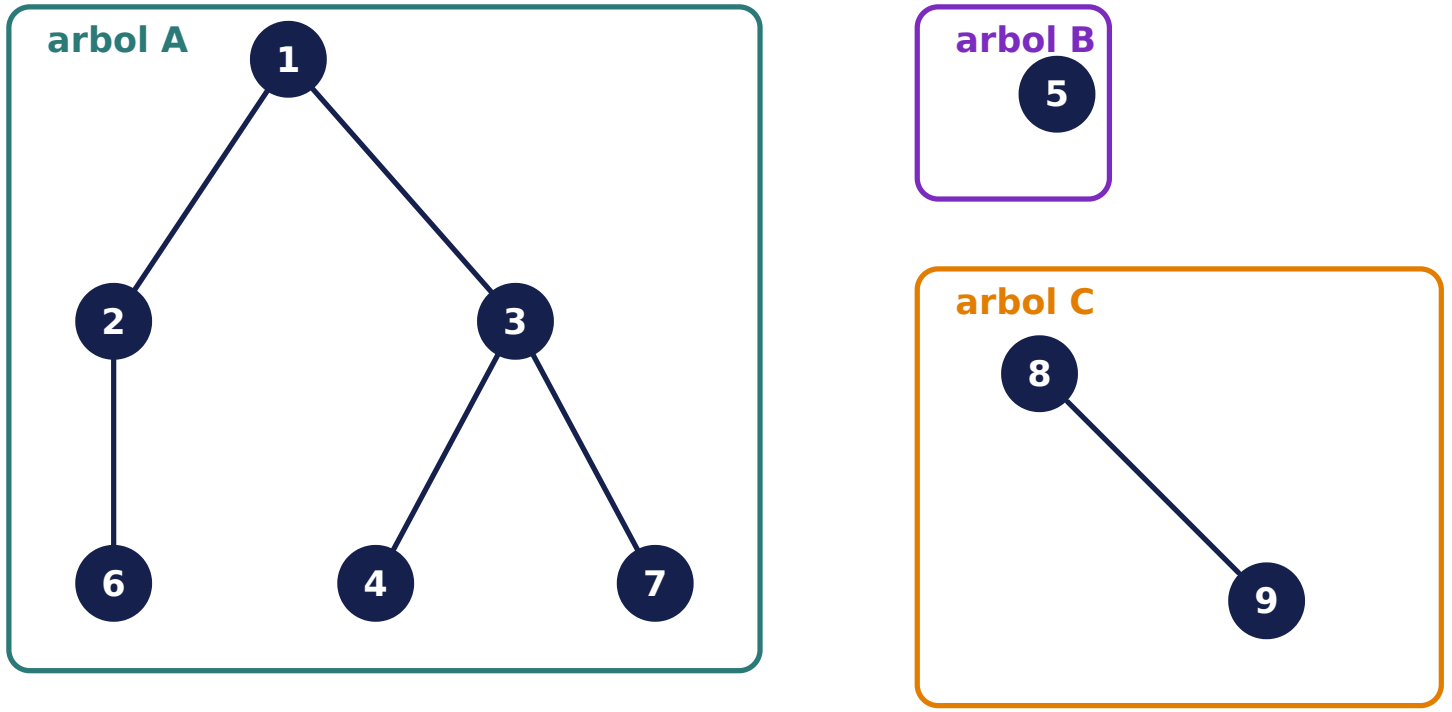
- G es árbol.
- $\forall\, u, v \in V$ existe un **único** camino entre ellos.
- G es conexo y $|E| = |V| - 1$.
- G no tiene ciclos y $|E| = |V| - 1$.

Árbol — ejemplo



Bosque

Cuando G solo cumple con **no tener ciclos** (pero puede ser disconexo), lo llamamos **bosque**: una colección de árboles.



Bloque 2 · Recorridos

1 · Fundamentos

2 · Recorridos

Grafos dirigidos (digrafos)

Casi todo lo que vimos aplica a los digrafos, **excepto** las definiciones de conexidad, árboles y bosques (que se vuelven un poco más finas).

La diferencia clave: las aristas ahora **tienen dirección**.

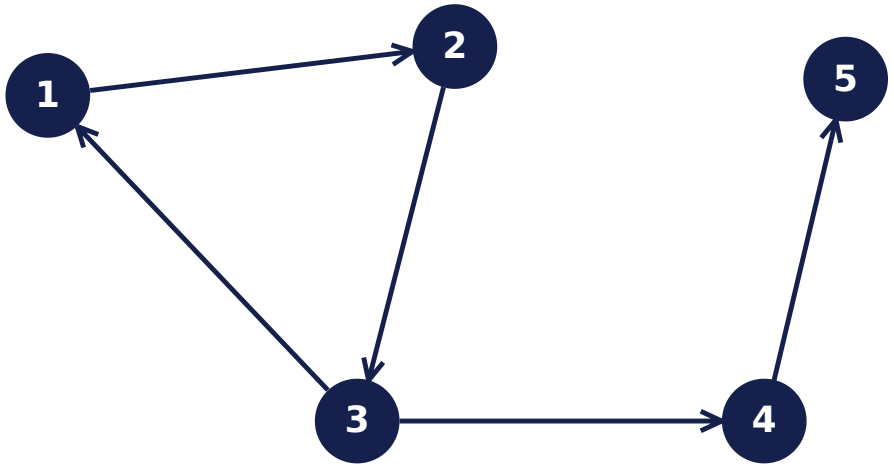
Definición

Un **digrafo** es un par ordenado $D = (V, A)$, donde:

- V es un conjunto finito y no vacío de **vértices**.
- $A \subseteq V \times V$ es un conjunto de **pares ordenados** llamados **arcos**.

$(u, v) \neq (v, u)$: el arco va **de** u **hacia** v.

Ejemplo de digrafo



Fíjate que $1 \to 2 \to 3 \to 1$ forma un ciclo dirigido, y $3 \to 4 \to 5$ es una "cola".

Matriz y lista de adyacencia para digrafos

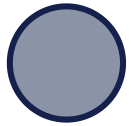
```cpp
int n, m;
cin >> n >> m;
vector<vector<int>> mtz_ady(n, vector<int>(n, 0));
vector<vector<int>> lst_ady(n);
for (int e = 0; e < m; ++e) {
    int u, v;
    cin >> u >> v;
    u--; v--;
    // A es un conjunto de pares ORDENADOS:
    // agregamos el arco SOLO en la dirección dada
    mtz_ady[u][v] = 1;
    lst_ady[u].push_back(v);
}
```

DFS — Depth First Search

DFS: la idea

Avanzar **lo más profundo posible**; cuando no se puede seguir, retroceder al vértice más cercano que aún tenga caminos sin explorar. Usamos **3 estados**:

No visitado — todavía no lo alcanzamos.

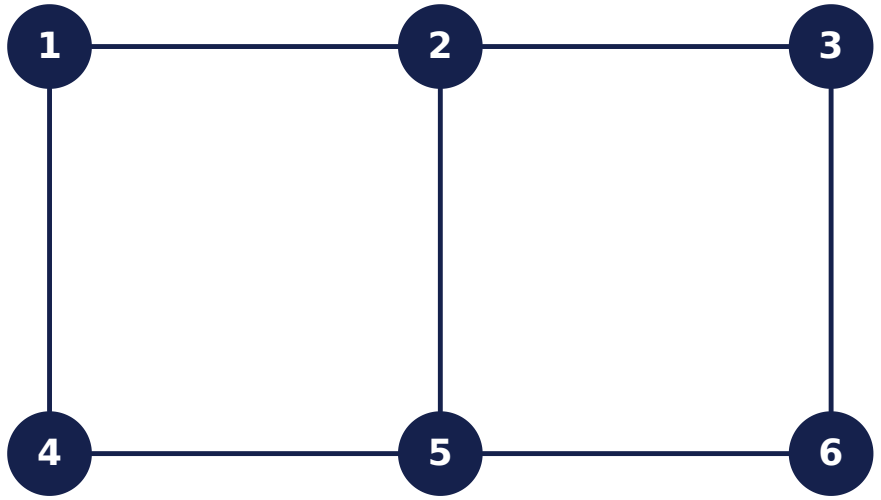


Parcialmente visitado — entré, pero aún no termino con él.

Totalmente visitado — ya exploré todo lo suyo.

Esta forma estandar se puede aplicar en muchos problemas de diferentes maneras

DFS — ejemplo en vivo (¿en qué orden visito?)



DFS — solución paso a paso

○ no visitado · ◉ parcialmente · ● totalmente · **anillo naranjo** = el que cambia

paso	1	2	3	4	5	6	pila (recursión)
t=1 gris(1)							1
t=2 gris(2)							1 2
t=3 gris(3)							1 2 3
t=4 gris(6)							1 2 3 6
t=5 gris(5)							1 2 3 6 5
t=6 gris(4)							1 2 3 6 5 4
t=7 negro(4)							1 2 3 6 5
t=8 negro(5)							1 2 3 6
t=9 negro(6)							1 2 3
t=10 negro(3)							1 2
t=11 negro(2)							1
t=12 negro(1)							(vacía)

DFS — implementación

```
enum Color { BLANCO, GRIS, NEGRO };     // no visitado / en proceso / terminado
vector<Color> color(n, BLANCO);

void dfs(int u) {
    color[u] = GRIS;                          // ENTRO a u: lo pinto gris
    for (int v : lst_ady[u]) {
        if      (color[v] == BLANCO) dfs(v);   // arista de árbol -> bajo por ella
        else if (color[v] == GRIS)   ;         // arista hacia atrás -> spoiler: ciclo
        else  /* color[v] == NEGRO */ ;        // ya cerramos ese vértice
    }
    color[u] = NEGRO;                         // SALGO de u: lo pinto negro
}

// para recorrer TODO el grafo (puede ser disconexo):
for (int u = 0; u < n; ++u)
    if (color[u] == BLANCO) dfs(u);
```

DFS: ¿qué información nos regala?

Con esos 3 estados y el orden de visita podemos:

- **Detectar ciclos** (encontrar un vecino ● **gris**, o sea, que todavía está en la pila de llamadas).
- Contar **componentes conexas**.
- Calcular tiempos de entrada/salida (orden topológico, puentes, etc. → *Grafos II*).

Cuidado con la recursión profunda: si V es grande, considera aumentar el stack o hacer DFS iterativo.

Ejercicio en vivo 1 — Building Roads

Hay n ciudades y m caminos. Queremos que se pueda viajar **entre cualquier par de ciudades**.

¿Cuál es el mínimo de caminos nuevos que hay que construir, y cuáles son?

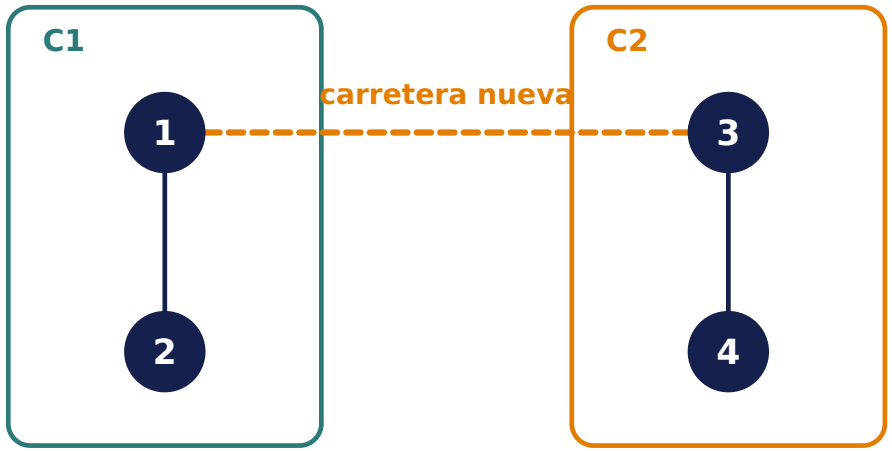
```
Entrada          Salida
4 2              1
1 2              2 3
3 4
```

CSES 1666 — Building Roads. Se acepta **cualquier** respuesta válida con el mínimo.

Paso 1: modelar

Antes de escribir código, traducir el enunciado a grafo:

- **ciudad** → vértice · **camino** → arista (no dirigida)
- "se puede viajar entre cualquier par" → el grafo debe quedar **conexo**



Paso 2: la idea

El grafo tiene **2 componentes** y bastó **1** carretera para unirlas.

Con k componentes, la respuesta siempre es $k - 1$: cada carretera nueva une dos componentes en una, así que hay que bajar de k a 1.

¿Y **cuáles** construir? Basta elegir **un representante** de cada componente y encadenarlos: $r_1 - r_2$, $r_2 - r_3$, ...

Entonces el problema se reduce a: encontrar las componentes conexas. Y eso ya sabemos hacerlo con DFS.

Paso 3: leer el grafo

```cpp
const int MAXN = 100005;            // n ≤ 1e5 según el enunciado

int n, m;
vector<int> ady[MAXN];              // MAXN listas de vecinos, todas vacías

int main() {
    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int a, b;
        cin >> a >> b;
        ady[a].push_back(b);
        ady[b].push_back(a);        // el camino sirve en ambos sentidos
    }
}
```

Acá usamos tamaño `n+1` y dejamos el índice 0 sin usar, así el vértice `i` vive en `ady[i]` y no hay que restar 1.

Paso 4: el DFS y el conteo

```
enum Color { BLANCO, GRIS, NEGRO };
Color color[MAXN];                          // global → arranca todo en 0 = BLANCO

void dfs(int u) {
    color[u] = GRIS;
    for (int v : ady[u])
        if (color[v] == BLANCO) dfs(v);
    color[u] = NEGRO;
}

// ... dentro de main(), después de leer:
vector<int> reps;
for (int u = 1; u <= n; ++u)
    if (color[u] == BLANCO) { reps.push_back(u); dfs(u); }
```

Paso 5: construir la respuesta

```cpp
int k = reps.size();                    // cantidad de componentes
cout << k - 1 << endl;                  // k - 1 carreteras

for (int i = 0; i + 1 < k; ++i)         // encadenamos representantes
    cout << reps[i] << " " << reps[i + 1] << endl;
```

Complejidad: $O(V + E)$. Cada vértice se pinta una vez y cada arista se mira dos veces (una por extremo).

Ojo con los casos borde: si el grafo ya es conexo, `reps.size() == 1` y se imprime `0` sin ninguna línea extra.

Solución completa

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100005;              // n ≤ 1e5 según el enunciado

int n, m;
vector<int> ady[MAXN];                // listas de adyacencia
enum Color { BLANCO, GRIS, NEGRO };
Color color[MAXN];                    // global → arranca todo en 0 = BLANCO

void dfs(int u) {                     // pinta TODA la componente de u
    color[u] = GRIS;
    for (int v : ady[u])
        if (color[v] == BLANCO) dfs(v);
    color[u] = NEGRO;
}

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);   // lectura rápida

    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int a, b;
        cin >> a >> b;
        ady[a].push_back(b);
        ady[b].push_back(a);          // el camino sirve en ambos sentidos
    }

    vector<int> reps;                 // un representante por componente
    for (int u = 1; u <= n; ++u)
        if (color[u] == BLANCO) {     // u no lo alcanzó ningún dfs anterior
            reps.push_back(u);        //   → inaugura una componente nueva
            dfs(u);                   //   → y la pintamos entera
        }

    int k = reps.size();              // cantidad de componentes
    cout << k - 1 << "\n";            // hacen falta k - 1 carreteras
    for (int i = 0; i + 1 < k; ++i)   // encadenamos los representantes
        cout << reps[i] << " " << reps[i + 1] << "\n";
}
```

Ejercicios para practicar

2 · Round Trip — ¿el grafo tiene un **ciclo**? Si lo tiene, imprimirlo.

Pista: es justo el caso "vecino GRIS" del DFS. Guarda el padre de cada vértice para reconstruir el ciclo.

3 · Counting Rooms — dado un mapa con `#` (pared) y `.` (piso), contar cuántas **habitaciones** hay.

Pista: es el mismo patrón de contar componentes, pero el grafo está escondido en la grilla.

Los tres comparten el mismo esqueleto: **recorrer todos los vértices y lanzar un DFS por cada uno sin visitar.** Lo que cambia es qué haces adentro.

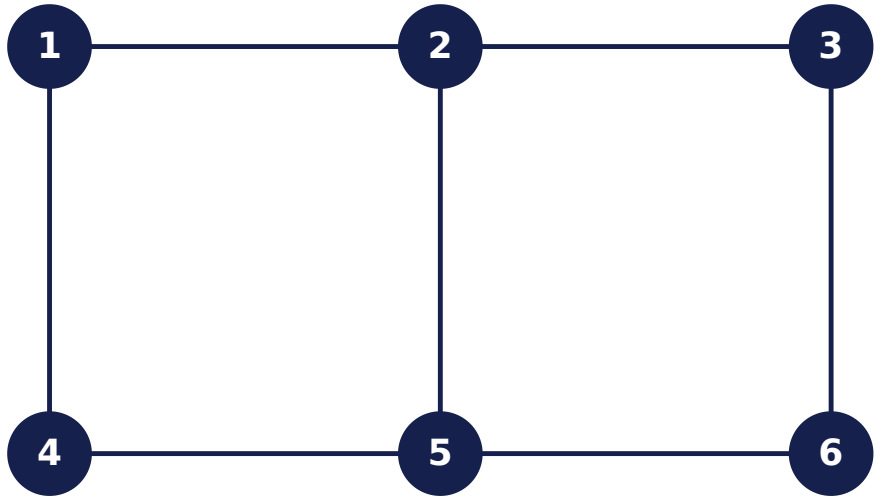
BFS — Breadth First Search

BFS: la idea

A partir de un vértice, **"inundamos"** a todos sus vecinos antes de avanzar al siguiente nivel. Se explora por **capas** (distancia 1, luego 2, luego 3...).

Se implementa con una **cola** (`queue`).

BFS — ejemplo en vivo (por capas)



BFS — solución paso a paso

○ no descubierto · ● en la cola · ● procesado · **anillo naranjo** = el que cambia

paso	1	2	3	4	5	6	cola
t=1 gris(1)							1
t=2 gris(2)							2
t=3 gris(4)							2 4
t=4 negro(1)							2 4
t=5 gris(3)							4 3
t=6 gris(5)							4 3 5
t=7 negro(2)							4 3 5
t=8 negro(4)							3 5
t=9 gris(6)							5 6
t=10 negro(3)							5 6
t=11 negro(5)							6
t=12 negro(6)							(vacía)

BFS — implementación

```
enum Color { BLANCO, GRIS, NEGRO };   // no descubierto / en la cola / procesado
vector<Color> color(n, BLANCO);

void bfs(int inicio) {
    queue<int> cola;
    color[inicio] = GRIS;                    // se pinta gris al encolar
    cola.push(inicio);
    while (!cola.empty()) {
        int u = cola.front(); cola.pop();
        for (int v : lst_ady[u]) {
            if (color[v] == BLANCO) {        // aún no lo descubrimos
                color[v] = GRIS;             // pasa a la frontera
                cola.push(v);
            }
        }
        color[u] = NEGRO;                    // ya miré todos sus vecinos
    }
}
```

Marcar al encolar (y no al desencolar) evita meter el mismo vértice varias veces.

Ejercicio en vivo 4 — Building Teams

Hay n alumnos y m amistades. Queremos dividirlos en **2 equipos**, de modo que **dos amigos nunca queden en el mismo equipo**.

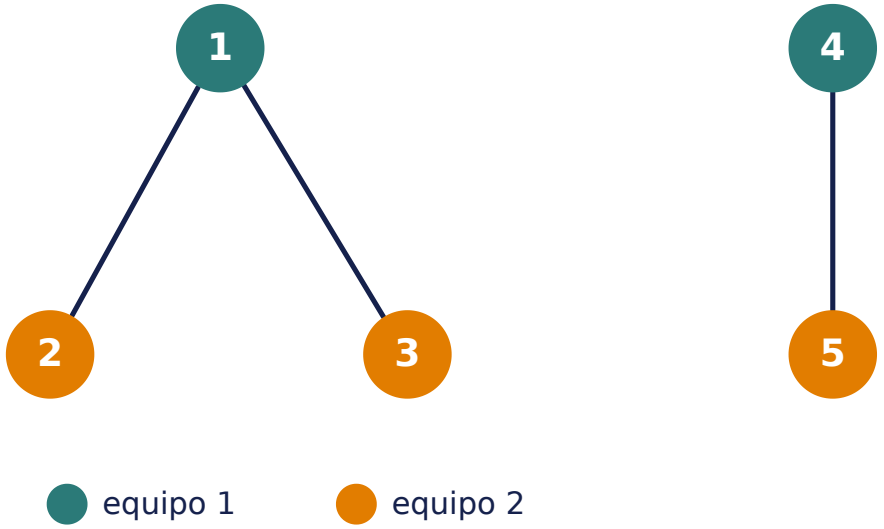
¿Se puede? Y si se puede, ¿cómo?

```
Entrada          Salida
5 3              1 2 2 1 2
1 2
1 3
4 5
```

CSES 1668 — Building Teams. Si no se puede, imprimir `IMPOSSIBLE`.

Paso 1: modelar

- **alumno** → vértice · **amistad** → arista
- "dos amigos en equipos distintos" → repartir los vértices en **2 grupos** de modo que ninguna arista una dos del mismo grupo



Paso 2: la idea

Recuerda que **BFS avanza por capas**. Si el vértice inicial es del equipo 1, entonces:

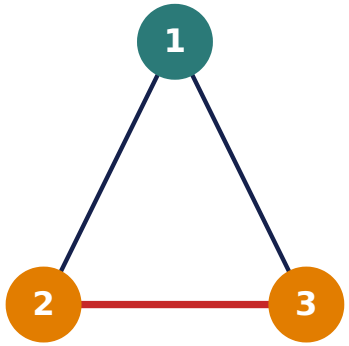
- sus vecinos (capa 1) → equipo **2**
- los vecinos de esos (capa 2) → equipo **1**
- ... y así, **alternando en cada capa**.

O sea: **la capa par va al equipo 1 y la capa impar al equipo 2.** No hay que decidir nada, el BFS ya nos entrega el orden.

Cada vez que descubro un vecino nuevo, le doy **el equipo contrario al mío**. Truco: si estoy en el equipo `e`, el contrario es `3 - e` (porque $3 - 1 = 2$ y $3 - 2 = 1$).

Paso 3: ¿cuándo es IMPOSSIBLE?

Si al mirar un vecino resulta que **ya fue descubierto y quedó en mi mismo equipo**, se acabó: no hay forma.



2 y 3 son amigos… y quedaron en el mismo equipo

Pasa exactamente cuando el grafo tiene un **ciclo de largo impar**: al dar la vuelta alternando, el color no cuadra al cerrar.

Paso 4: dos arreglos, dos preguntas distintas

Acá necesitamos guardar **dos cosas** por vértice, y conviene no mezclarlas:

Arreglo	¿Qué responde?	Valores
`color[]`	¿en qué **estado** del BFS está?	BLANCO / GRIS / NEGRO
`equipo[]`	¿en qué **equipo** quedó?	1 o 2

El `color[]` es el mismo de siempre: maneja el recorrido. El `equipo[]` es **la respuesta** que nos pide el problema. Son preguntas distintas, así que van separadas.

Paso 5: el BFS que asigna equipos

```
const int MAXN = 100005;
int n, m;
vector<int> ady[MAXN];
enum Color { BLANCO, GRIS, NEGRO };
Color color[MAXN];                      // estado del BFS
int equipo[MAXN];                       // la respuesta: 1 o 2

// ... dentro de main(), despues de leer el grafo:
for (int u = 1; u <= n; ++u) {
    if (color[u] != BLANCO) continue;  // ya lo recorrio un BFS anterior
    color[u] = GRIS;                    // arranco una componente nueva
    equipo[u] = 1;
    queue<int> cola;
    cola.push(u);
    while (!cola.empty()) {
        int x = cola.front(); cola.pop();
        for (int y : ady[x]) {
            if (color[y] == BLANCO) {          // vecino nuevo
                color[y] = GRIS;               //   -> a la frontera
                equipo[y] = 3 - equipo[x];     //   -> equipo contrario
                cola.push(y);
            } else if (equipo[y] == equipo[x]) {   // amigo en MI equipo
                cout << "IMPOSSIBLE\n";
                return 0;
            }
        }
        color[x] = NEGRO;                      // termine con x
    }
}
```

Solución completa

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100005;

int n, m;
vector<int> ady[MAXN];
enum Color { BLANCO, GRIS, NEGRO };
Color color[MAXN];                    // estado del BFS
int equipo[MAXN];                     // la respuesta: 1 o 2

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int a, b; cin >> a >> b;
        ady[a].push_back(b); ady[b].push_back(a);
    }

    for (int u = 1; u <= n; ++u) {
        if (color[u] != BLANCO) continue;
        color[u] = GRIS;
        equipo[u] = 1;
        queue<int> cola;
        cola.push(u);
        while (!cola.empty()) {
            int x = cola.front(); cola.pop();
            for (int y : ady[x]) {
                if (color[y] == BLANCO) {
                    color[y] = GRIS;
                    equipo[y] = 3 - equipo[x];
                    cola.push(y);
                } else if (equipo[y] == equipo[x]) {
                    cout << "IMPOSSIBLE\n";
                    return 0;
                }
            }
            color[x] = NEGRO;
        }
    }
    for (int u = 1; u <= n; ++u) cout << equipo[u] << " \n"[u == n];
}
```

Lo que hay que notar

Es **el mismo esqueleto de siempre**: recorrer todos los vértices y lanzar un recorrido por cada uno sin visitar. Lo único que cambia es qué hacemos adentro.

- El grafo puede ser **disconexo** → por eso el `for` externo sobre todos los alumnos.
- El `else if` cubre a los vecinos **grises y negros** por igual: si ya lo descubrimos, basta comparar equipos.
- **Complejidad:** $O(V + E)$, igual que un BFS normal.

A un grafo que se puede repartir así en 2 grupos se le llama **bipartito**. Aparece harto en ProgComp.

Complejidad de DFS y BFS

	Matriz	Lista
DFS	$O(V^2)$	$O(V+E)$
BFS	$O(V^2)$	$O(V+E)$

Con **lista** ambos son $O(V+E)$: visitamos cada vértice una vez y cada arista una vez. Con **matriz** pagamos $O(V)$ por vértice para hallar sus vecinos $\rightarrow O(V^2)$.

Distancias en grafos

Distancia en un grafo

Muchísimos problemas piden la **distancia mínima** para ir de un lugar a otro.

Partimos por el caso en que **cada arista cuesta 1**. Ahí BFS es perfecto: como explora por capas, la primera vez que llega a un vértice lo hace por el camino **más corto**.

Distancia mínima con BFS (aristas de peso 1)

```
vector<int> dist(n, -1);          // -1 = no visitado

void bfs(int inicio) {
    dist[inicio] = 0;
    queue<int> cola; cola.push(inicio);
    while (!cola.empty()) {
        int u = cola.front(); cola.pop();
        for (int v : lst_ady[u]) {
            if (dist[v] == -1) {          // primera vez que lo alcanzo
                dist[v] = dist[u] + 1;    // → distancia mínima
                cola.push(v);
            }
        }
    }
}
```

Cierre

Lo que viene (Grafos II)

Con DFS, BFS ya tienes una base sólida. Lo próximo:

- **Dijkstra** y caminos mínimos con pesos arbitrarios
- Bellman-Ford y Floyd-Warshall
- BFS 0-1

Para practicar

- **CSES Problem Set** → sección *Graph Algorithms* (ideal para partir).
- **Codeforces** → filtrar por tags `dfs and similar`, `graphs`, `shortest paths`.
- **AtCoder** → los problemas ABC C/D suelen tener grafos muy limpios.

Consejo: implementa DFS y BFS **de memoria** hasta que te salgan sin pensar. Esa base se usa en casi todo.

¿Preguntas?