Grafos I: From zero to (0.3)·hero

V. Benjamín Letelier Lazo
(vletelier@udec.cl ∨ benjamin.letelier@udc.es)

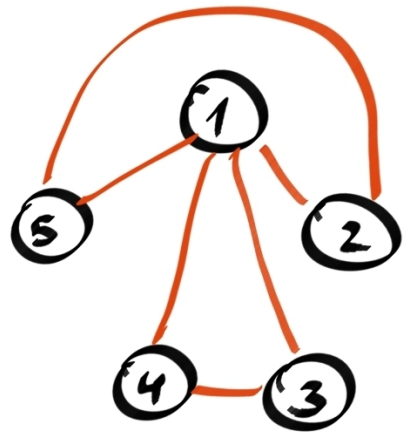
- Los grafos son un modelo matemático ampliamente utilizado.
- Permiten modelar conexiones e interacciones entre distintos entes de un sistema.

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

- V es un conjunto finito y no vacío, cuyos elementos llamamos *vértices*.
- $E = \{ \{a, b\} : \ a, b \in V \}$, cuyos elementos llamamos *aristas*.

Fome, veamos ejemplos mejor

G (modelo visual)



$G = (V, E)$

$V = \{1, 2, 3, 4, 5\}$

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

G (modelo ProgComp)

|V| → |E|

```
5 6
1 2 ----> E1
5 2 ----> E2
3 4
1 3
1 5
1 4 ----> E6
```

¿Cómo leo un grafo en ProgComp?

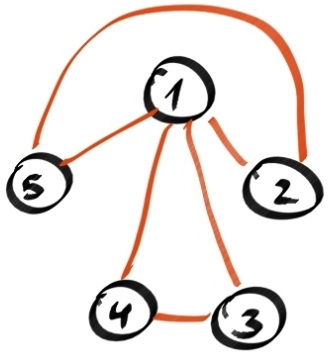
```
//n = |V|, m = |E|.
int n, m;
cin >> n >> m;
for(int e = 0; e < m; ++e) {
    //E_e es un elemento de E, tal que E_e = (u,v).
    int u, v;
    cin >> u >> v;
}
```

¿Y cómo los almaceno estas cosas?

Usando matrices de adyacendia o listas de adyacencia.

Matriz de adyacencia

G (modelo visual)



$G = (V, E)$

$V = \{1, 2, 3, 4, 5\}$

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

G (modelo ProgComp)

|V| → 5 6 ← |E|

1 2 ⟶ E_1
5 2 ⟶ E_2
3 4
1 3
1 5
1 4 ⟶ E_6

G (matriz de adyacencia)

	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

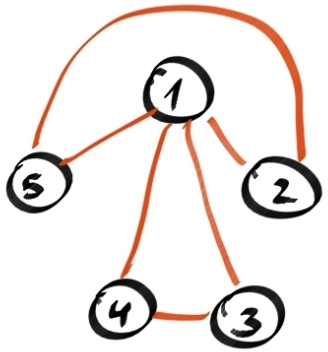
0: no existe arista
1: sí existe arista

¿Cómo creo una matriz de adyacencia?

```cpp
//n = |V|, m = |E|.
int n, m;
cin >> n >> m;
vector<vector<int>> mtz_ady(n, vector<int>(n, 0));
for(int e = 0; e < m; ++e) {
    //E_e es un elemento de E, tal que E_e = (u,v).
    int u, v;
    cin >> u >> v;
    //Los valores u,v toman normalmente los valores [1..n].
    //si es el caso restamos 1 para que queden en [0..n-1].
    u--; v--;
    //Como E es un conjunto de pares no ordenados, añadimos la arista en ambas direcciones.
    mtz_ady[u][v] = 1;
    mtz_ady[v][u] = 1;
}
```

Lista de adyacencia

G (modelo visual)



$G = (V, E)$

$V = \{1, 2, 3, 4, 5\}$

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

G (modelo ProgComp)

|V| → 5 → 6 ← |E|

1 2 ⟶ E_1

5 2 ⟶ E_2

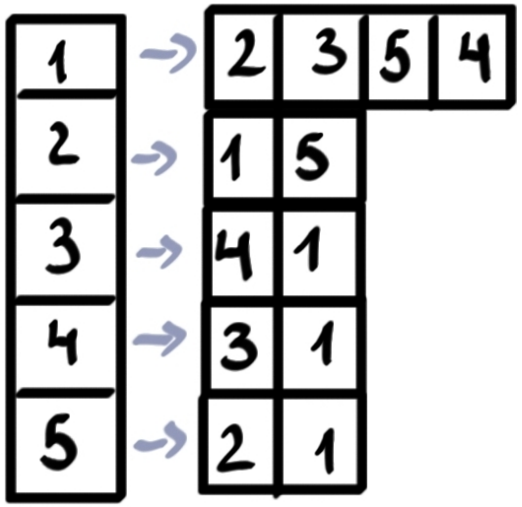
3 4

1 3

1 5

1 4 ⟶ E_6

G (lista de adyacencia)



¿Cómo creo una lista de adyacencia?

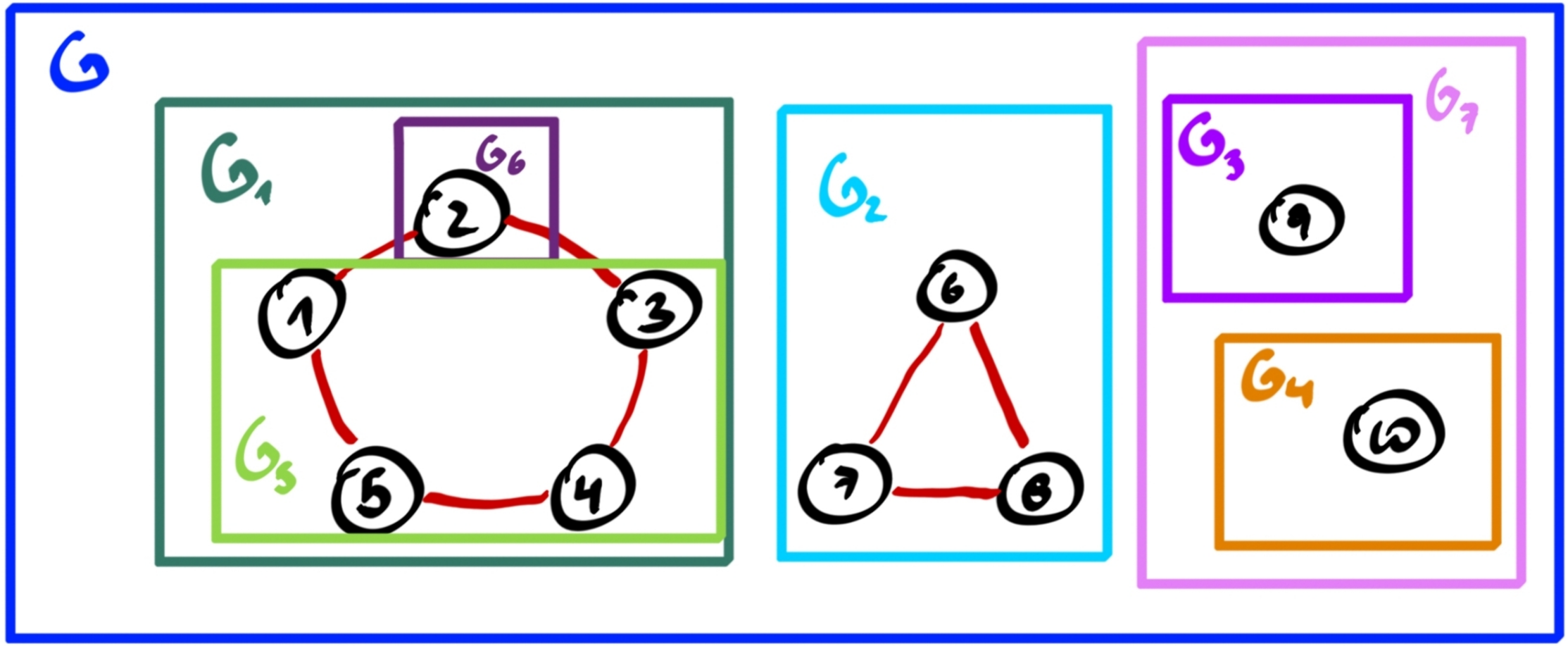
```cpp
//n = |V|, m = |E|.
int n, m;
cin >> n >> m;
vector<vector<int>> lst_ady(n);
for(int e = 0; e < m; ++e) {
    //E_e es un elemento de E, tal que E_e = (u,v).
    int u, v;
    cin >> u >> v;
    //Los valores u,v toman normalmente los valores [1..n].
    //si es el caso restamos 1 para que queden en [0..n-1].
    u--; v--;
    //Como E es un conjunto de pares no ordenados, añadimos la arista en ambas direcciones.
    lst_ady[u].push_back(v);
    lst_ady[v].push_back(u);
}
```

¿Cuáles son las ventajas entre usar matriz y lista de adyacencia?

	Matriz	Lista
Creación de la estructura	$O(n^2)$	$O(n+m)$
¿Existe la arista (u, v)?	$O(1)$	$O(m)$
¿Cuántos vecinos tiene el vértice v?	$O(n)$	$O(1)$
¿Cuáles son los vecinos del vértice v?	$O(n)$	$O(m)$

Representemos el siguiente grafo (ejemplo en vivo 1)

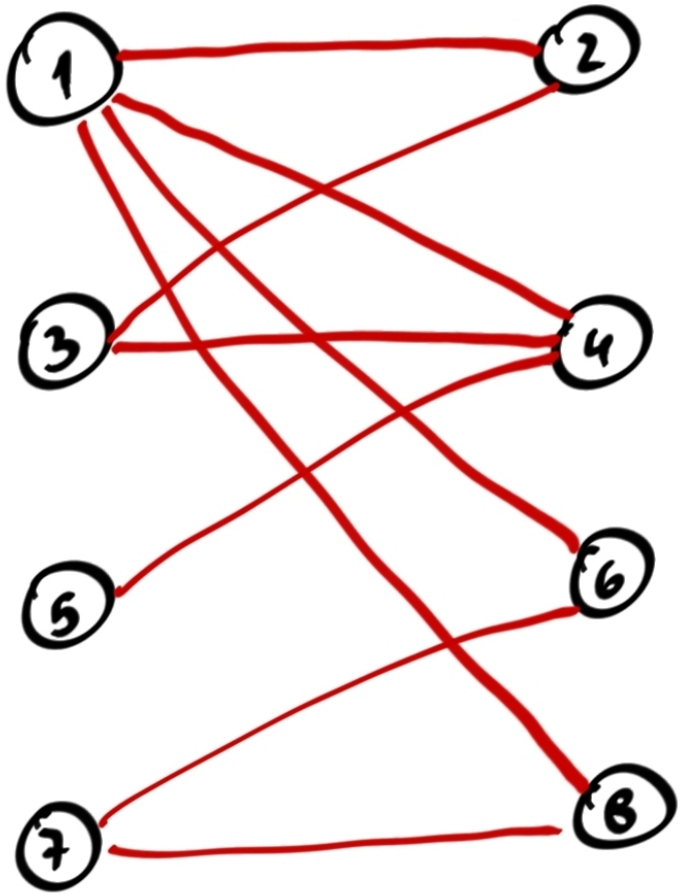
```
10 8
1 2
3 2
3 4
6 8
1 5
5 4
7 8
6 7
```


G
G1
G6
2
1
3
G5
5
4
G2
6
7
8
G7
G3
9
G4
10

Definiremos el *recorrido* desde el vértice v_0 hasta v_l en G, como una secuencia ordenada de vértices de G, tal que $\forall j \in \{0, \ldots, l-1\}, \{v_j, v_{j+1}\} \in E$.

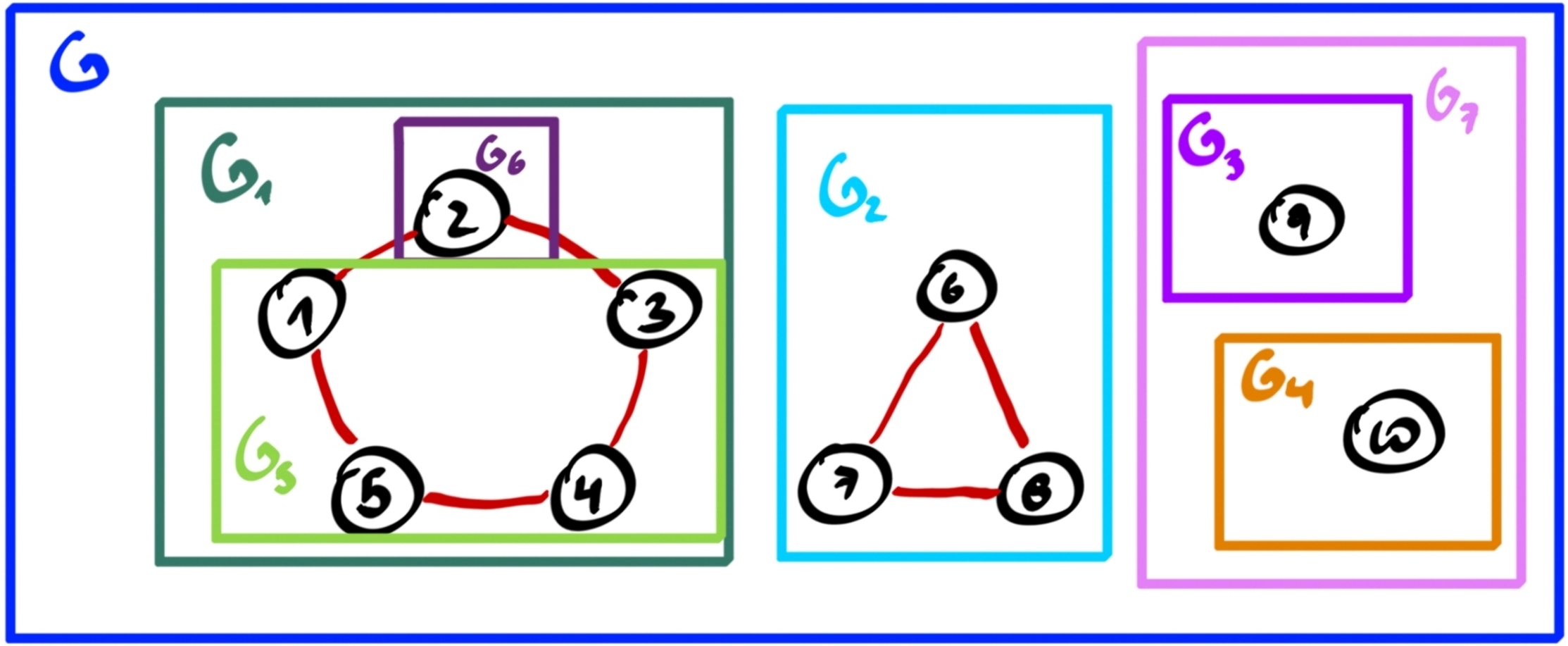
- Recorrido $R = (v_0, v_1, \ldots, v_{l-1}, v_l)$, con un largo $|R| = l$.
- Si el **recorrido** es tal que $v_0 = v_l$, lo denominaremos *circuito*.
- Si el **recorrido** no contiene vértices repetidos, lo denomiraremos *camino*.
- Si el **circuito** no contiene vértices repetidos (excepto por v_0 y v_l), lo denomiraremos *ciclo*.

Mucho texto, veamos ejemplos (ejemplo en vivo 2)

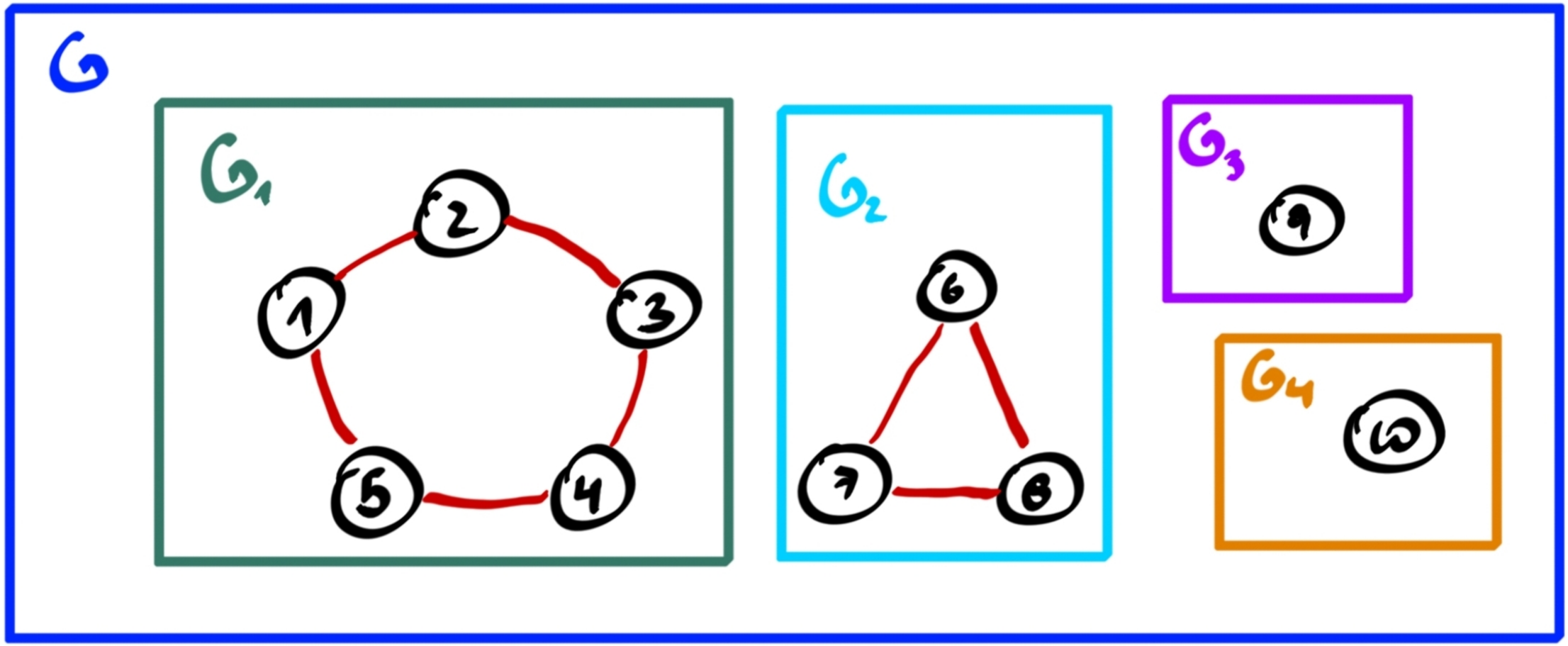


- Un grafo se dice *conexo* si es *trivial* o $\forall u, v \in V$ existe un camino en G.
- Si un grafo no es conexo, se dice *disconexo*.
- Una *componente conexa* de G es un subgrafo maximal en la propiedad de conexidad, es decir, no existe otro subgrafo conexo conteniendo estrictamente a alguna componente conexa.

Retomando el ejemplo 1



Eliminamos subgrafos no maximales del ejemplo 1



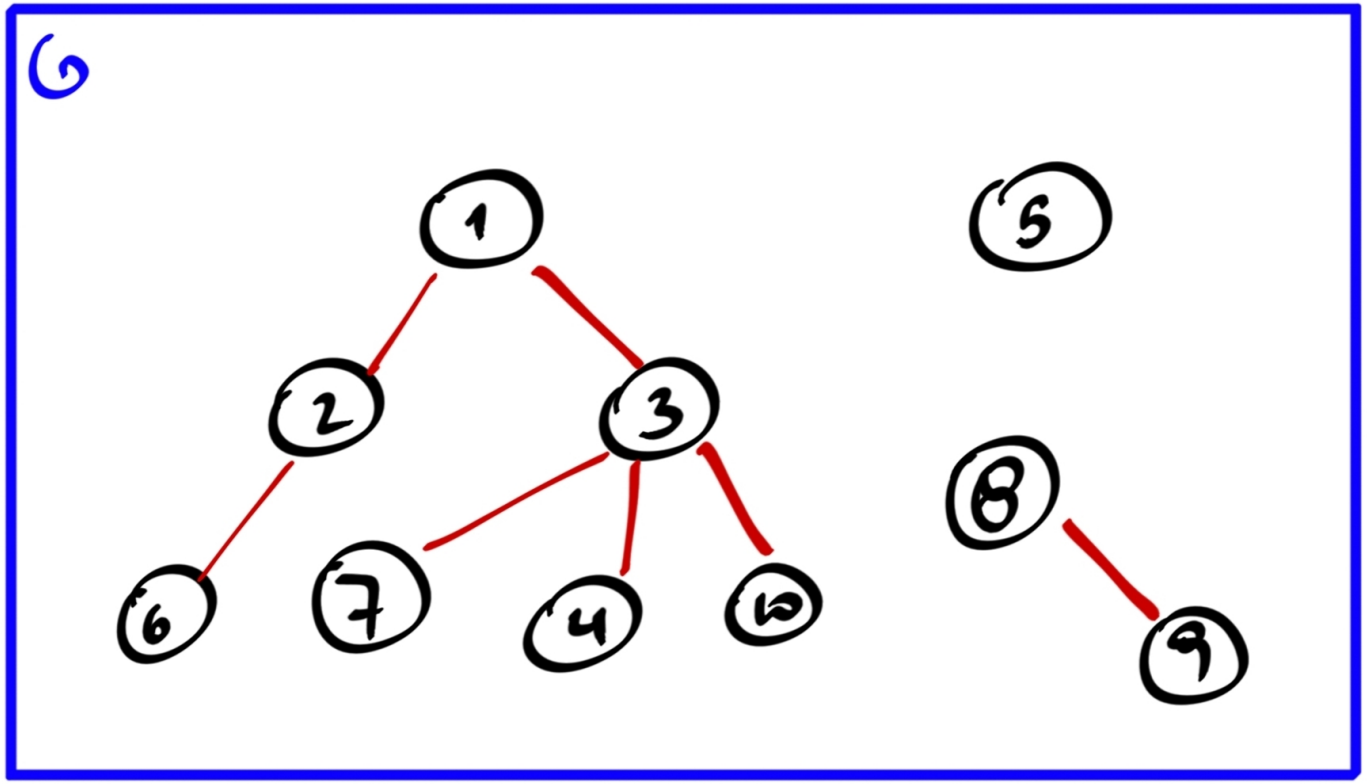
- G es **disconexo**, pero posee **4 componentes conexas**.
- Cada **componente conexa es un subgrafo conexo maximal** de G.
- G_3 y G_4 son **grafos triviales** (tienen un único vértice y ninguna arista).
- G_2 es el **grafo completo** K_3 (cada vértice tiene $\left|G_2\right| - 1$ vecinos).
- G_1 y G_3 son grafos 2**-regular** (cada vértice tiene exactamente 2 vecinos).

Si un grafo G es conexo y sin ciclos, entonces diremos que G es un *árbol*.
Entonces, las siguientes proposiciones son equivalentes:

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

No entendí, ejemplo please (ejemplo en vivo 3)

Cuándo un grafo G únicamente cumpla con no poseer ciclos, lo llamaremos *bosque*.

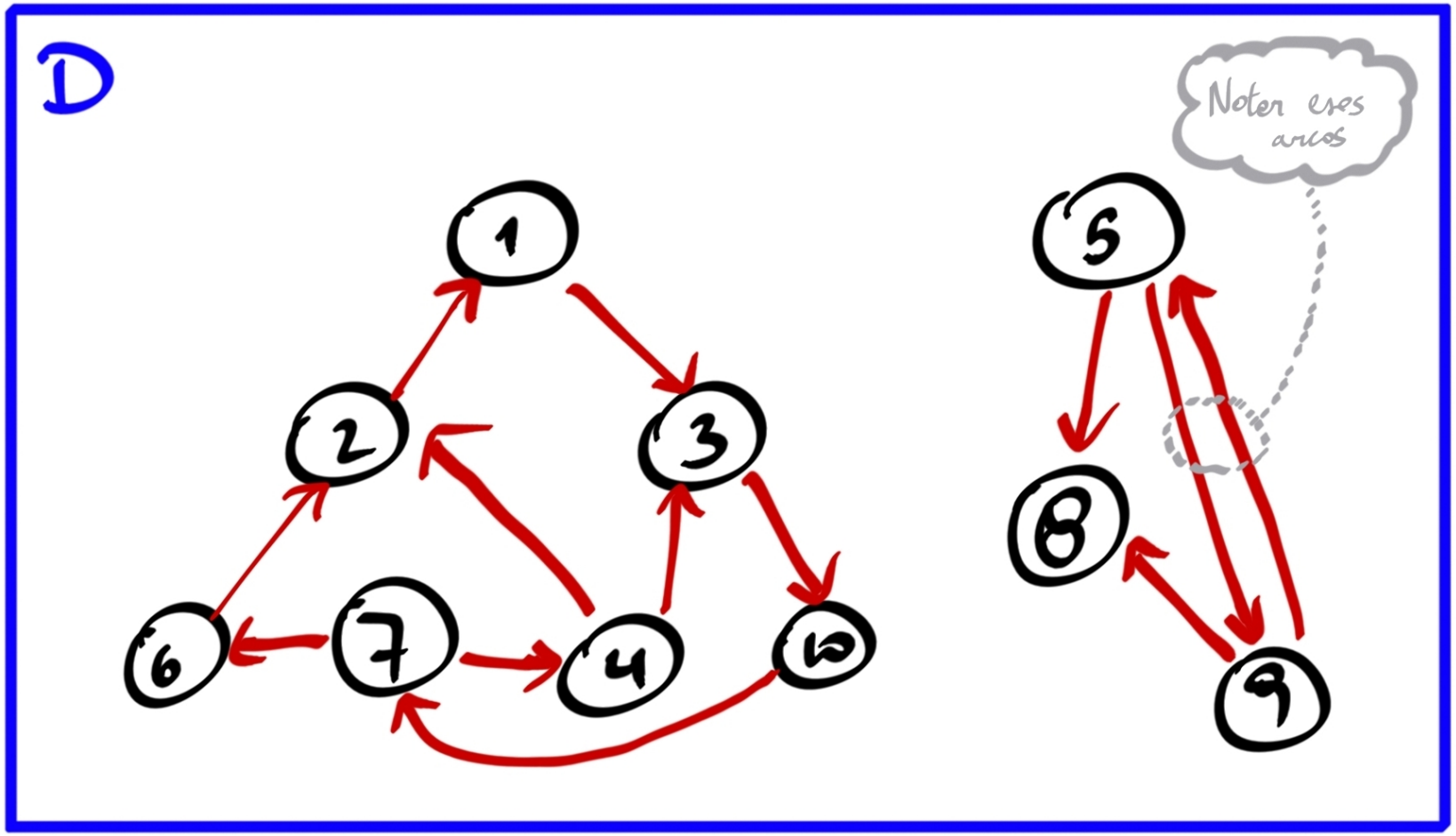


Casi todo lo que vimos aplica a los digrafos, excepto las definiciones de conexidad, árboles y bosques.

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

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

Ejemplo de digrafo



Código de matriz y lista de adyacencia para digrafos

```cpp
int n, m;
cin >> n >> m;
vector<vector<int>> mtx_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;
    //Los valores u,v toman normalmente los valores [1..n].
    //si es el caso restamos 1 para que queden en [0..n-1].
    u--; v--;
    //Como E es un conjunto de pares ordenados, añadimos el arco en la dirección dada.
    //matriz de adyacencia
    mtx_ady[u][v] = 1;
    //lista de adyacencia
    lst_ady[u].push_back(v);
}
//Pregunta para ustedes: ¿Se puede usar una matriz y una lista de adyacencia a la vez?
```

La idea principal es avanzar lo máximo posible hacia adelante y cuando no podemos seguir avanzando, retrocedemos hasta el vértice más cercano a nuestra posición que tenga un camino que no haya sido recorrido. Utilizaremos 3 estados para diferenciar los vértices.

¡¡¡Este algoritmo entrega muchísima información del grafo!!!

Recorrido DFS en un grafo (ejemplo en vivo 4)

Código para realizar DFS sobre un grafo/digrafo con lista de adyacencia

```
int NO_VISITADO = 0, PARCIALMENTE_VISITADO = 1, TOTALMENTE_VISITADO = 2;
vector<int> estado(n, NO_VISITADO);
void dfs(int vertice) {
    estado[vertice] = PARCIALMENTE_VISITADO;
    for(auto &vecino: lst_ady[vertice]) {
        if(color[vecino] == NO_VISITADO)
            dfs(vecino); //Vértice que no ha sido explorado aún.
        else if(estado[vecino] == PARCIALMENTE_VISITADO)
            //Vecino que vuelve a otro vértice ya explorado, pero no por completo.
        else if(estado[vecino] == TOTALMENTE_VISITADO)
            //Vecino que vuelve a otro vértice ya explorado por completo.
    }
    estado[vertice] = TOTALMENTE_VISITADO;
}
```

Resolviendo este ejercicio usando DFS

Resolviendo estos dos ejercicios usando DFS: Ejercicio 2 y Ejercicio 3.

La idea principal es, a partir de un vértice, "inundar" a todos sus vecinos antes de pasar a la siguiente iteración.

Recorrido BFS en un grafo (ejemplo en vivo 5)

Código para realizar BFS sobre un grafo/digrafo con lista de adyacencia

```
int NO_VISITADO = 0, PARCIALMENTE_VISITADO = 1, TOTALMENTE_VISITADO = 2;
vector<int> estado(n, NO_VISITADO);
void bfs(int vertice) {
    queue<int> cola; cola.push(vertice);
    while(!cola.empty()) {
        int enfrente = cola.front();
        cola.pop();
        estado[enfrente] = PARCIALMENTE_VISITADO;
        for(auto vecino: lst_ady[enfrente]) {
            if(estado[vecino] == NO_VISITADO)
                cola.push(vecino);
        }
        estado[enfrente] = TOTALMENTE_VISITADO;
    }
}
```

Resolviendo este ejercicio usando BFS

¿Cuál es la complejidad de usar DFS o BFS?

	Matriz	Lista
DFS	$O(n^2)$	$O(n+m)$
BFS	$O(n^2)$	$O(n+m)$

Es común encontrarse ejercicios donde tenemos encontrar la distancia mínima para ir de un lugar a otro.

Primero comenzaremos cuando la distancia de moverse sobre una arista es igual a 1. Luego, finalizaremos la clase con el caso general.

Código para la distancia mínima sobre un grafo/digrafo con lista de adyacencia usando BFS

```
vector<int> distancia(n, 1'000'000'000);
void bfs(int vertice) {
    distancia[vertice] = 0;
    queue<int> cola; cola.push(vertice);
    while(!cola.empty()) {
        int enfrente = cola.front();
        cola.pop();
        for(auto vecino: lst_ady[enfrente]) {
            if(distancia[vecino] > distancia[enfrente] + 1) {
                distancia[vecino] = distancia[enfrente] + 1;
                cola.push(vecino);
            }
        }
    }
}
```

Resolviendo este ejercicio usando BFS

¿Qué hacemos si la distancia de la arista varía?

Se debe aplicar una técnica similar a las ya vistas, pero primero hay que saber como almacenar esto en una matriz y lista de adyacencia. A estos grafos y digrafos que poseen diferentes pesos/distancias en las aristas, los llamaremos *ponderados*.

Matriz de adyacencia para un grafo ponderado

```cpp
//n = |V|, m = |E|.
int n, m;
cin >> n >> m;
//NO_EXISTE reemplaza al anterior 0 y debe ser mayor que el max valor de todos los w.
int NO_EXISTE = 1'000'000'007;
vector<vector<int>> mtz_ady(n, vector<int>(n, NO_EXISTE));
for(int e = 0; e < m; ++e) {
    //u, v son aristas y w es el peso/distancia.
    int u, v, w;
    cin >> u >> v >> w;
    //Comentar si u,v toman valores entre [0..n-1].
    u--; v--;
    mtz_ady[u][v] = w;
    //Comentar si es digrafo.
    mtz_ady[v][u] = w;
}
```

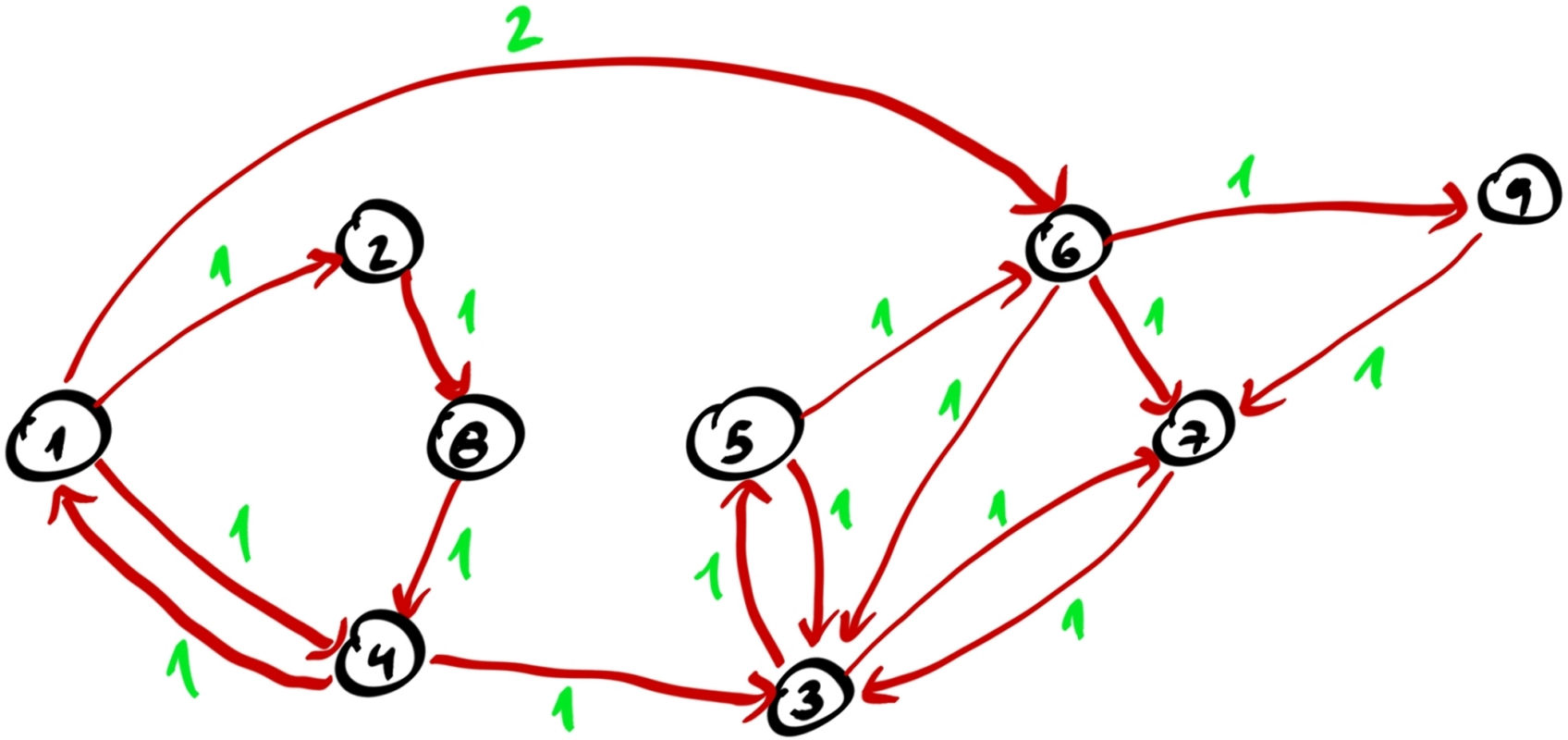
Lista de adyacencia para un grafo ponderado

```cpp
//n = |V|, m = |E|.
int n, m;
cin >> n >> m;
vector<vector<pair<int, int>>> lst_ady(n);
for(int e = 0; e < m; ++e) {
    //u, v son aristas y w es el peso/distancia.
    int u, v, w;
    cin >> u >> v >> w;
    //Comentar si u,v toman valores entre [0..n-1].
    u--; v--;
    lst_ady[u].push_back({v,w});
    //Comentar si es digrafo.
    lst_ady[v].push_back({u,w});
}
```

Sirve para la determinación del camino más corto **desde un vértice de origen al resto de vértices** de un grafo o digrafo ponderado. Se base en una estrategia *greedy*.

Antes de programarlo, pensemos como podría funcionar basándonos en el caso particular cuando el peso/distancia en cada arista era 1.

Cuál es el camino mínimo en este digrafo, iniciando en el vértice 1 (ejemplo en vivo 6)



Código Dijkstra sobre un grafo/digrafo ponderado con lista de adyacencia

```
vector<int> distancia(n, 1'000'000'000);
void dijkstra(int vertice) {
    distancia[vertice] = 0
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> min_heap;
    min_heap.push({distancia[vertice], vertice});
    while(!min_heap.empty()) {
        auto [distancia_enfrente, enfrente] = min_heap.top();
        min_heap.pop();
        if(distancia_enfrente > distancia[enfrente]) continue;
        for(auto [vecino, peso_vecino]: lst_ady[enfrente]) {
            if(distancia[vecino] > distancia_enfrente + peso_vecino) {
                distancia[vecino] = distancia_enfrente + peso_vecino;
                min_heap.push({distancia[vecino], vecino});
            }
        }
    }
}
```

Complejidad

	Matriz	Lista
Dijkstra	$O(n^2 \cdot log_2(n))$	$O((n+m) \cdot log_2(n))$

Ejercicio en vivo 6

Resolviendo este ejercicio usando Dijsktra

¿Siempre debo usar Dijkstra?

Dijkstra no puede usarse cuando existen pesos negativos.

¿Por qué?

- Se rompe la invariante con la que aseguramos la correctitud del procedimiento.

¿Qué hacer en ese caso?

- Usar Bellman-Ford o Floyd-Warshall (**si el grafo no posee ciclos negativos**).