CURSO DE PROGRAMACIÓN COMPETITIVA

GRAFOS

CURSO DE PROGRAMACIÓN COMPETITIVA URJC - 2025

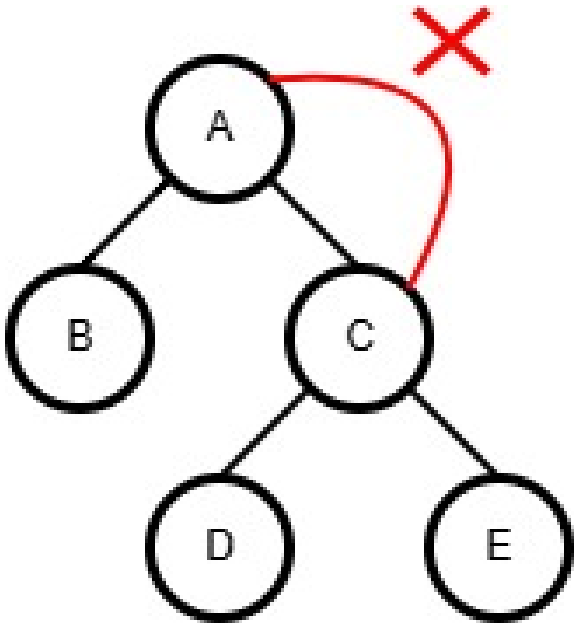
Organizadores:

- Isaac Lozano (isaac.lozano@urjc.es)
- Sergio Salazar (sergio.salazar@urjc.es)
- **Adaya Ruiz** (am.ruiz.2020@alumnos.urjc.es)
- Eva Gómez (e.gomezf.2020@alumnos.urjc.es)
- Lucas Martín (lucas.martin@urjc.es)
- Iván Penedo (ivan.penedo@urjc.es)
- Alicia Pina (alicia.pina@urjc.es)
- Sara García (sara.garciar@urjc.es)
- Raúl Fauste (r.fauste.2020@alumnos.urjc.es)
- Alejandro Mayoral (a.mayoralg.2020@alumnos.urjc.es)
- **David Orna** (de.orna.2020@alumnos.urjc.es)

Grafos

Árboles: representan relaciones de jerarquía

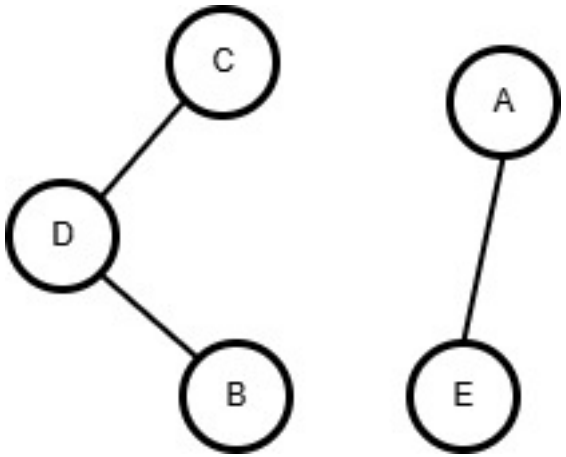
- Tienen un padre (menos raíz)
- Pueden tener hijos
- No admiten ciclos!

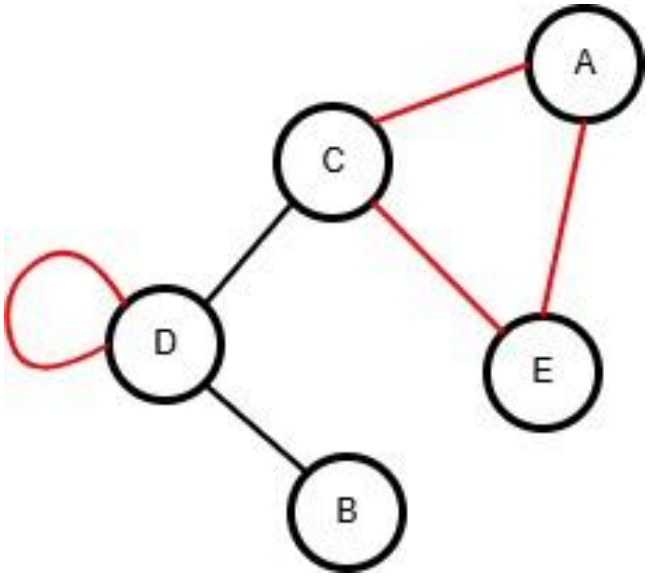


Grafos

Grafos: más libertad para representar un sistema

- Permite ciclos
- Permite bucles sobre un mismo elemento
- Permite grupos aislados

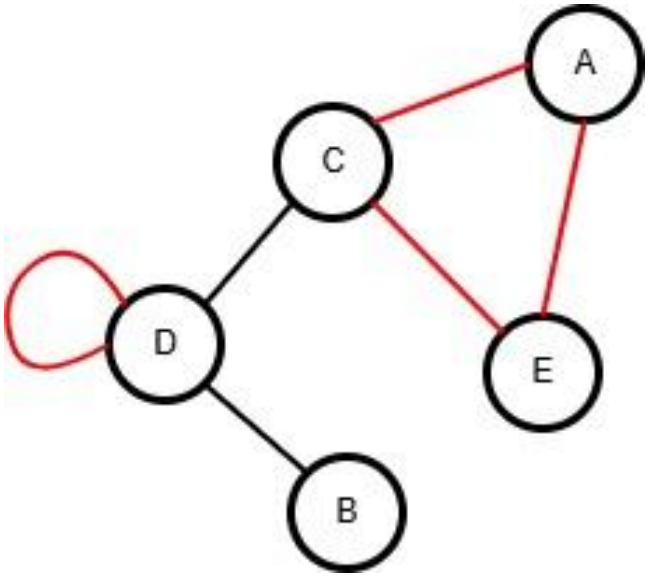




Definición

Grafo → conjunto de vértices V y aristas E que los relacionan tal que G=(V,E)

- **Vértices**: V={A,B,C,D,E}
- **Aristas**: E={(A,C),(A,E),(B,D),(C,D),(C,E),(D,D)}



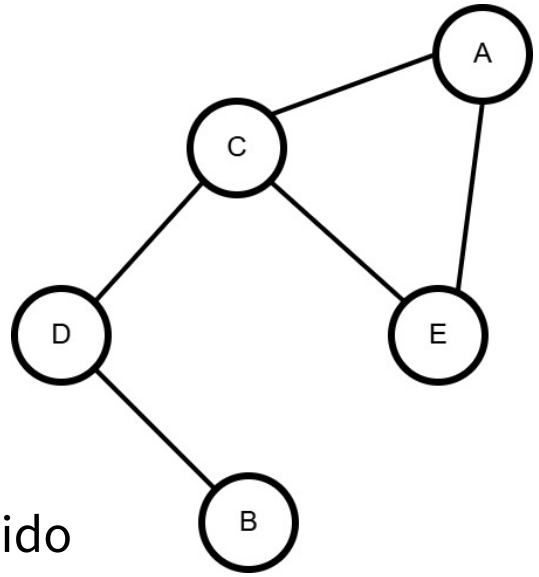
Dirigidos o no dirigidos

No dirigido:

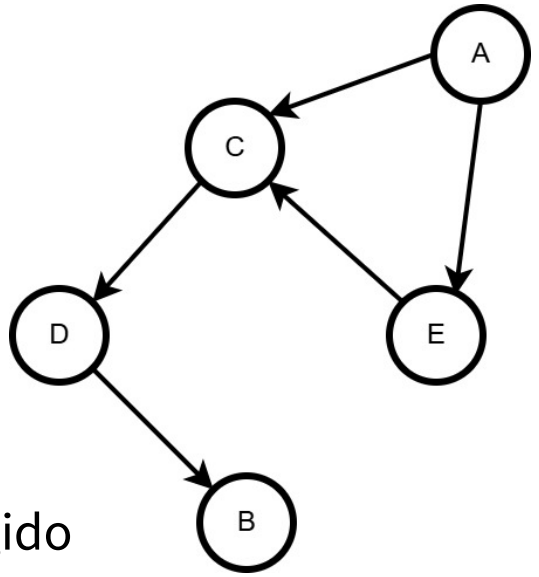
E={(A,C),(C,A),(A,E),(E,A),(B,D),(D,B),(C,D),(D,C),(C,E),(E,C)}

Dirigido:

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



No dirigido

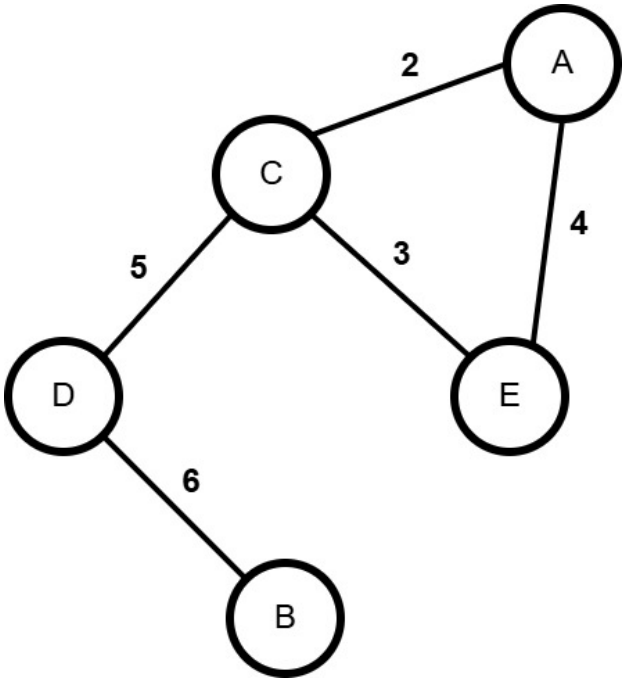


Dirigido

Ponderados

Grafos ponderados: las aristas tienen pesos/valores

E={(A,C,2),(A,E,4),(B,D,6),(C,D,5),(C,E,3)}



Representación – matriz de adyacencia

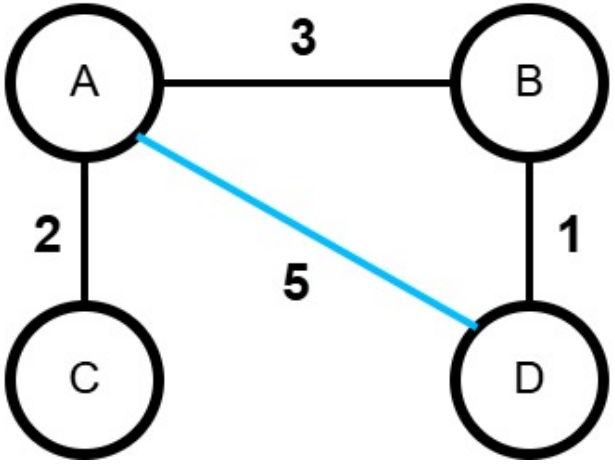
- Array de dos dimensiones

- 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

- Uso: grafos densos(N~=5000)

Representación – matriz de adyacencia

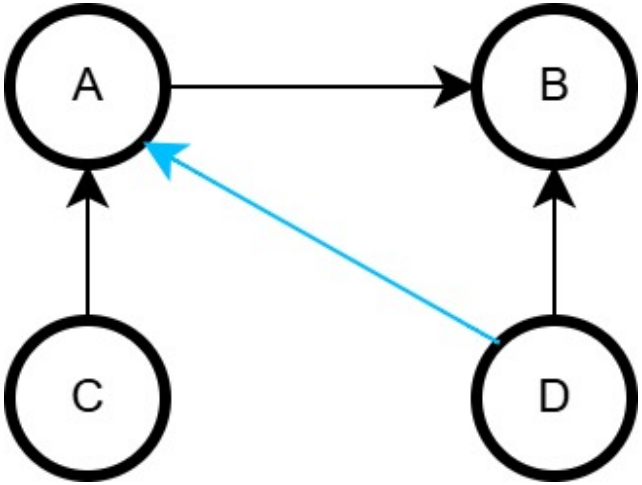
Es simétrica para grafos no dirigidos → **m[A][D] = m[D][A]**



V	A	B	C	D
A	0	3	2	5
B	3	0	0	1
C	2	0	0	0
D	5	1	0	0

Representación – matriz de adyacencia

NO simétrica para grafos dirigidos → m[A][D] = **0**, m[D][A] = **1**



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

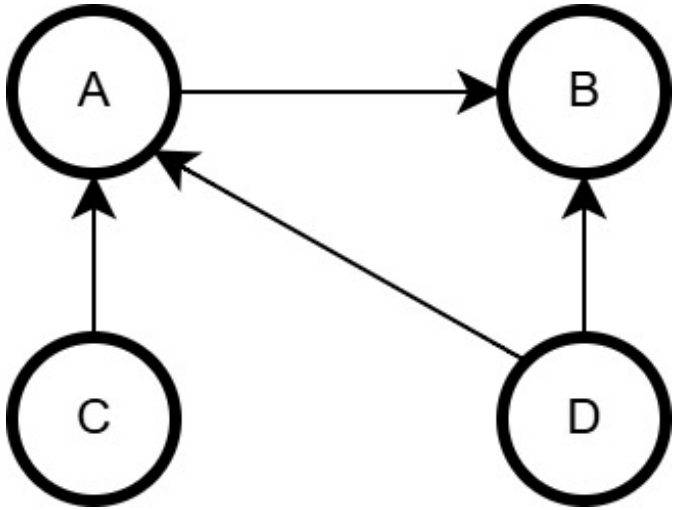
Representación – lista de adyacencia

- Array con lista de aristas

- 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

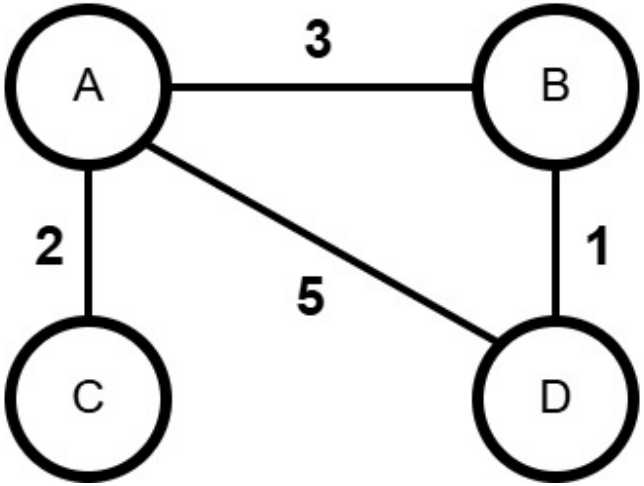
- Útil para grafos dispersos

Representación – lista de adyacencia



A	→	{B}
B	→	{}
C	→	{A}
D	→	{A,B}

Representación – lista de adyacencia



A	→	{(B,3),(C,2),(D,5)}
B	→	{(A,3), (D,1)}
C	→	{(A,2)}
D	→	{(A,3),(B,1)}

Implementación – input

- Información que nos dan:
  - Número de nodos
  - Número de aristas
  - Listado de aristas
  - Dirigido? → enunciado

```
4 4
A B 3
A D 3
A C 2
B D 1
```

```
4 4
A B
C A
D A
D B
```

Implementación – matriz de adyacencia

```
n, m = map(int, input().strip().split())
matriz = [[0]*n for _ in range(n)]

for _ in range(m):
        v1, v2, a = map(int, input().strip().split())
        m[v1][v2] = a
        m [v2][v1] = a     → PARA NO DIRIGIDOS
```

- Si no nos dan el valor usaremos 1 en vez de a!

Implementación – lista de adyacencia

```
n, m = map(int, input().strip().split())
g = [[] for _ in range(n)]

for _ in range(m):
    v1, v2, a = map(int, input().strip().split())
    g[v1].append((v2,a))
    g[v2].append((v1,a))   → PARA NO DIRIGIDOS
```

- Si no nos dan el valor usaremos 1 en vez de a!

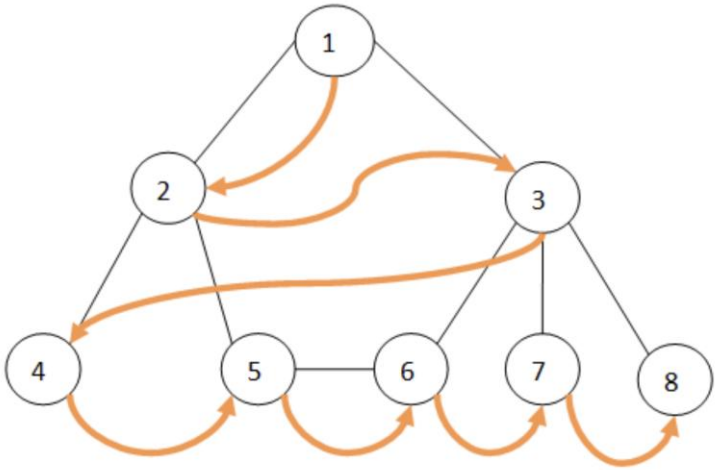
Implementación – otras opciones

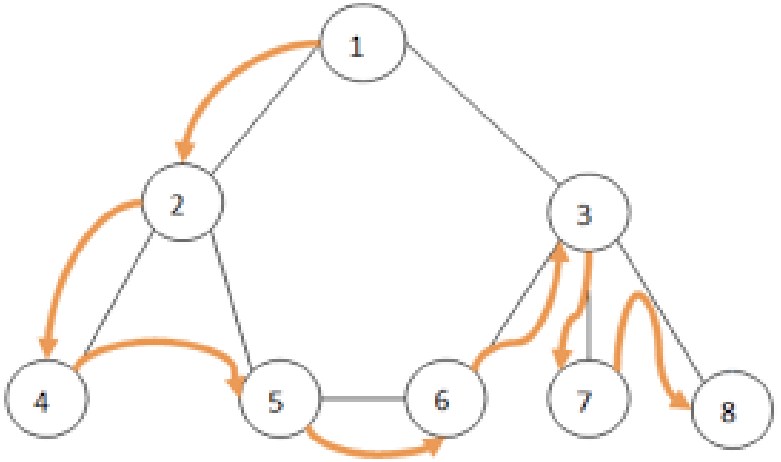
- **Mapas**: permite utilizar cualquier etiqueta para los vertices, no solo enteros!
  - Problema del clasificatorio para el Ada Byron

- **Lista de aristas**: otra implementación para representar grafos, algoritmos específicos

Recorridos

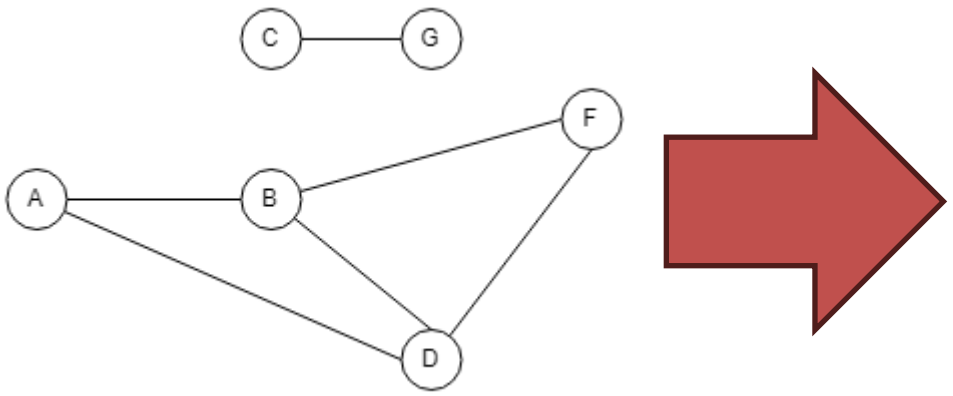
- Recorrer todos los vertices en un grafo
  - Recorrido en anchura (BFS)
  - Recorrido en profundida (DFS)





Recorridos

- Se comienzan desde un vértice inicial
- ¿Y si no se puede recorrer todo el grafo?



Empezar de nuevo el recorrido desde un nodo que no hayamos visitado!!

BFS

- Recorrido por niveles
- Implementado con una cola y un array o set de visitados

```
def bfs(grafo, visitados,origen):
    cola=deque()
    cola.append(origen)
    while cola:
        aux=cola.popleft()
        if aux not in visitados:
            visitados.add(aux)
        for adj in grafo[aux]:
            if adj not in visitados:
                cola.append()
```

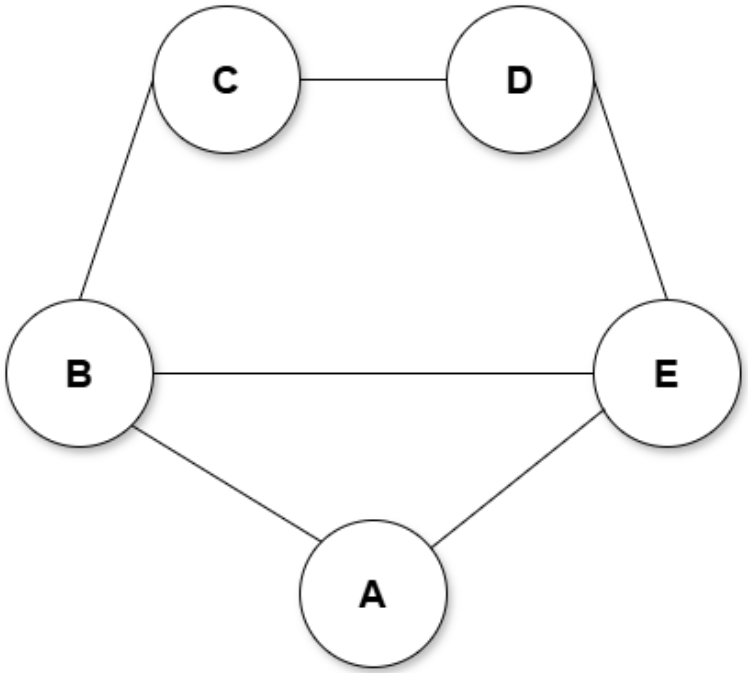
- Desde el vértice inicial recorremos los vecinos
- Marcamos los visitados

BFS

Cola:

Visitados:

A	B	C	D	E



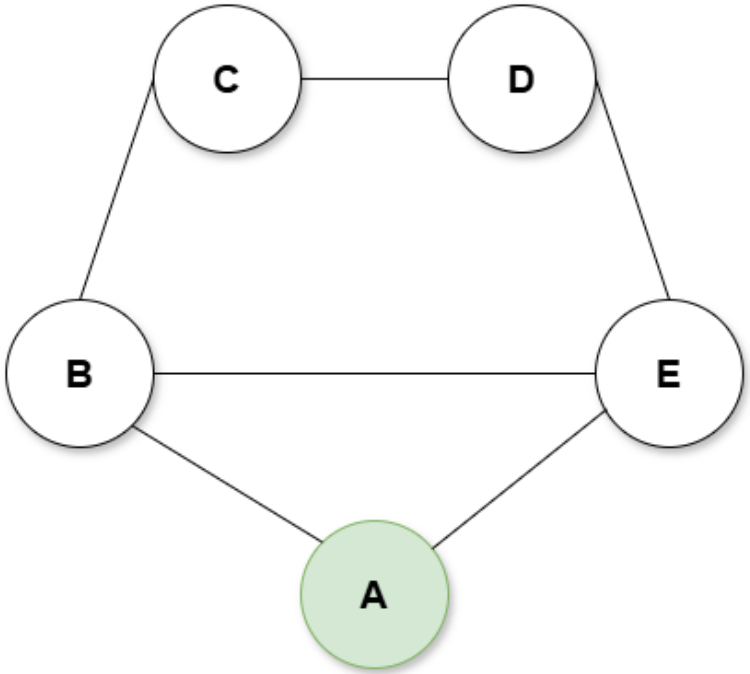
BFS

Cola:

A				

Visitados:

A	B	C	D	E



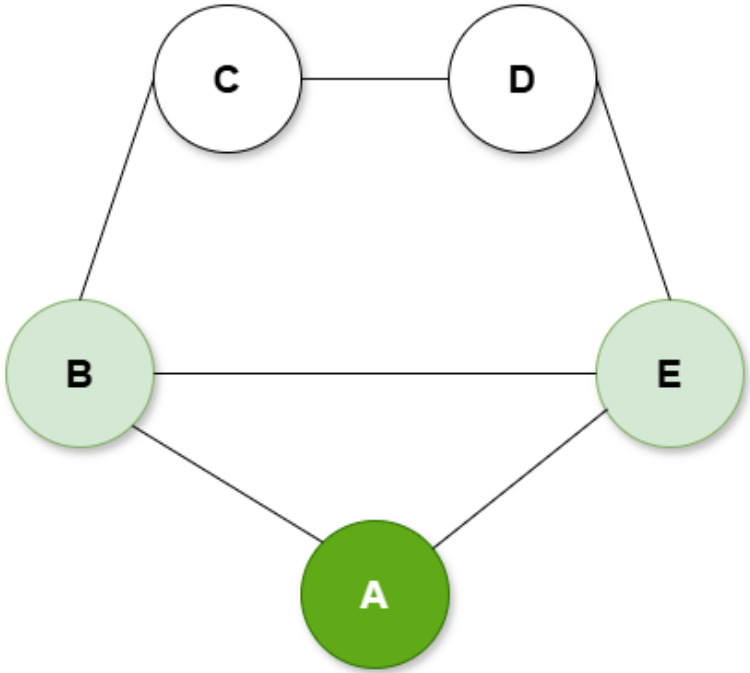
BFS

Cola:

A	B	E		

Visitados:

A	B	C	D	E



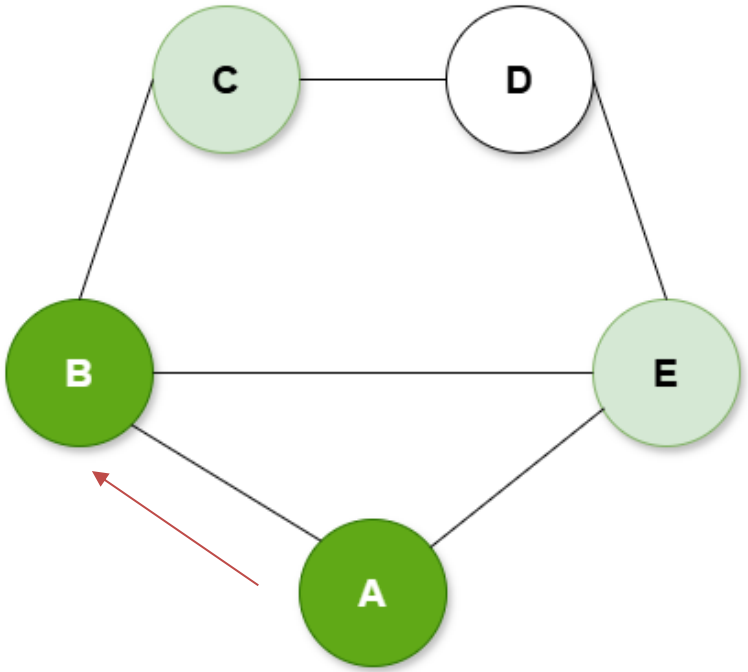
BFS

Cola:

B	E	C		

Visitados:

A	B	C	D	E



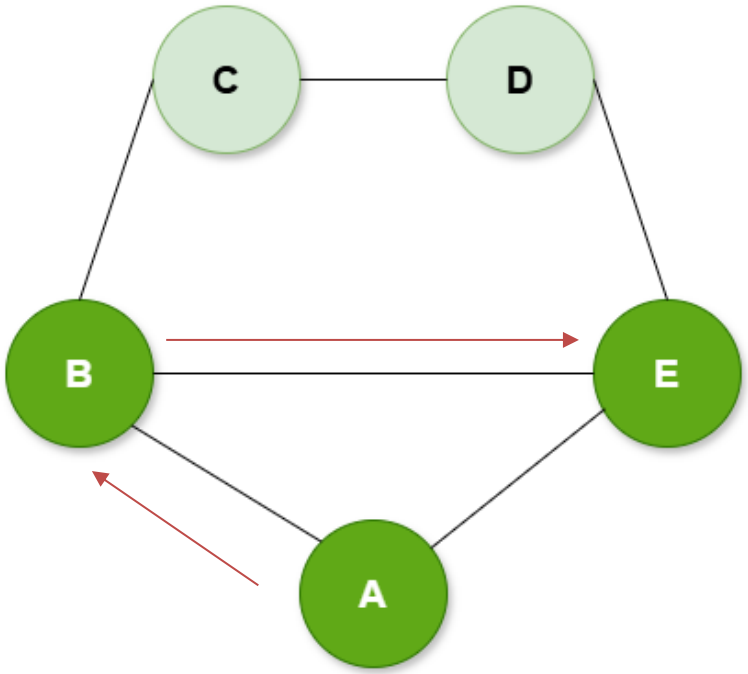
BFS

Cola:

E	C	D		

Visitados:

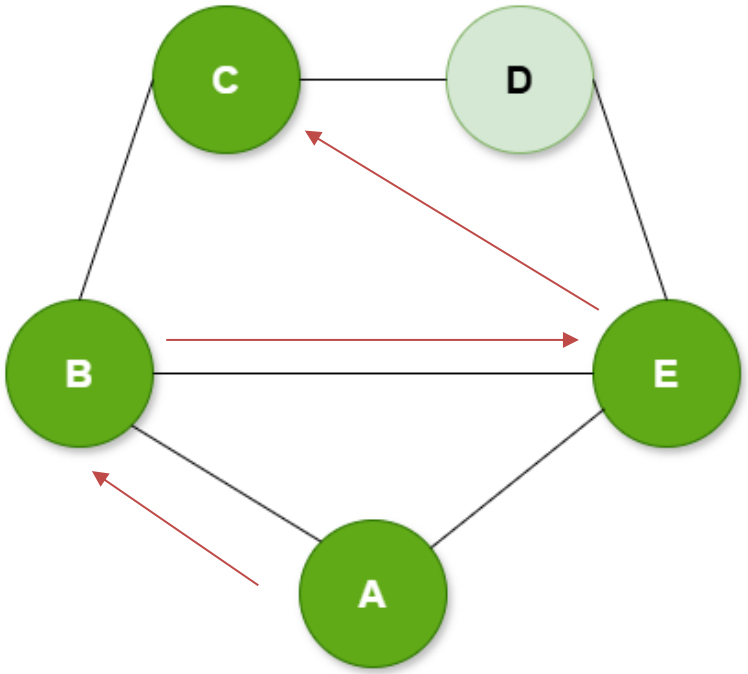
A	B	C	D	E



BFS

Cola:

Visitados:



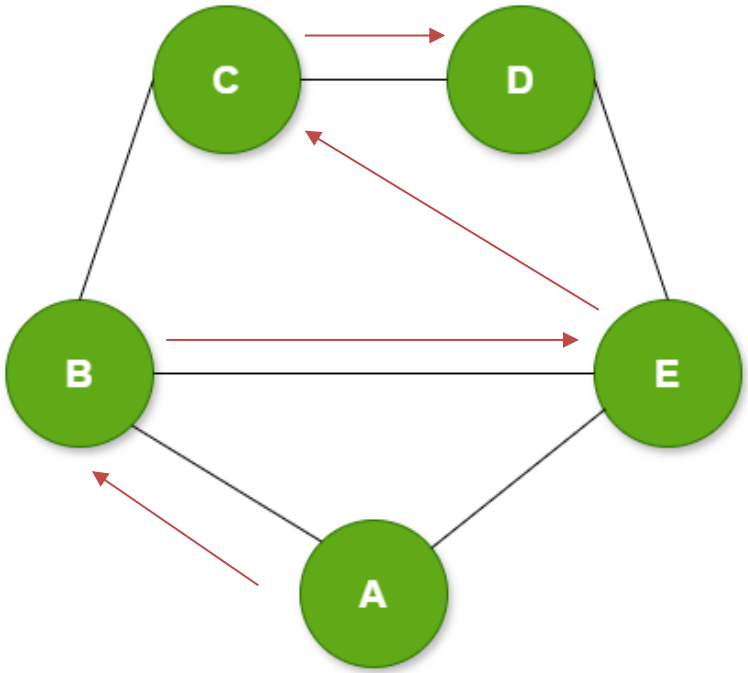
BFS

Cola:

D				

Visitados:

A	B	C	D	E



BFS - Ejercicio

Problema propuesto: https://open.kattis.com/problems/modulosolitaire

Entrada:

- s_0: inicio
- m
- Secuencia de operadores

$$s_j = (s_{j-1} \cdot a_{i_j} + b_{i_j}) \bmod m.$$

Salida:
Mínimo de operaciones necesarias para llegar a 0

Ejemplo:

- $s_0 = 1$

i	a	b
1	2	1
2	3	1

BFS

Problema propuesto: https://open.kattis.com/problems/modulosolitaire

Entrada:

- s_0: inicio
- m
- Secuencia de operadores

$$s_j = (s_{j-1} \cdot a_{i_j} + b_{i_j}) \bmod m.$$

Salida:
Operaciones necesarias para llegar a 0

Ejemplo:

- $s_0 = 1$

i	a	b
1	2	1
2	3	1

1

BFS

Problema propuesto: https://open.kattis.com/problems/modulosolitaire

Entrada:

- s_0: inicio
- m
- Secuencia de operadores

$$s_j = (s_{j-1} \cdot a_{i_i} + b_{i_i}) \bmod m.$$

Salida:
Operaciones necesarias para llegar a 0

Ejemplo:

- $s_0 = 1$

i	a	b
1	2	1
2	3	1

$$1 \xrightarrow[i=1]{(1 \cdot 2 + 1) mod\ 5} 3$$

BFS

Problema propuesto: https://open.kattis.com/problems/modulosolitaire

Entrada:

- s_0: inicio
- m
- Secuencia de operadores

$$s_j = (s_{j-1} \cdot a_{i_j} + b_{i_j}) \bmod m.$$

Salida:
Operaciones necesarias para llegar a 0

Ejemplo:

- $s_0 = 1$

i	a	b
1	2	1
2	3	1

$$1 \xrightarrow[i=1]{(1 \cdot 2 + 1) mod\ 5} 3 \xrightarrow[i=2]{(3 \cdot 3 + 1) mod\ 5} 0$$

BFS

¿Y si queremos guardarnos cuántos niveles visitamos?

BFS

¿Y si queremos guardarnos cuántos niveles visitamos?

```
def bfs_niveles(grafo, visitados,origen):
    cola=deque()
    cola.append(origen)
    cola.append(None)
    niveles=0
    while cola:
        aux=cola.popleft()
        if aux is None:
            if cola:
                niveles+=1
                cola.append(None)
        else:
            if aux not in visitados:
                visitados.add(aux)
            for adj in reversed(grafo[aux]):
                if adj not in visitados:
                    cola.append()
```

Elemento fuera del dominio que marca el final de un nivel e inicio del siguiente

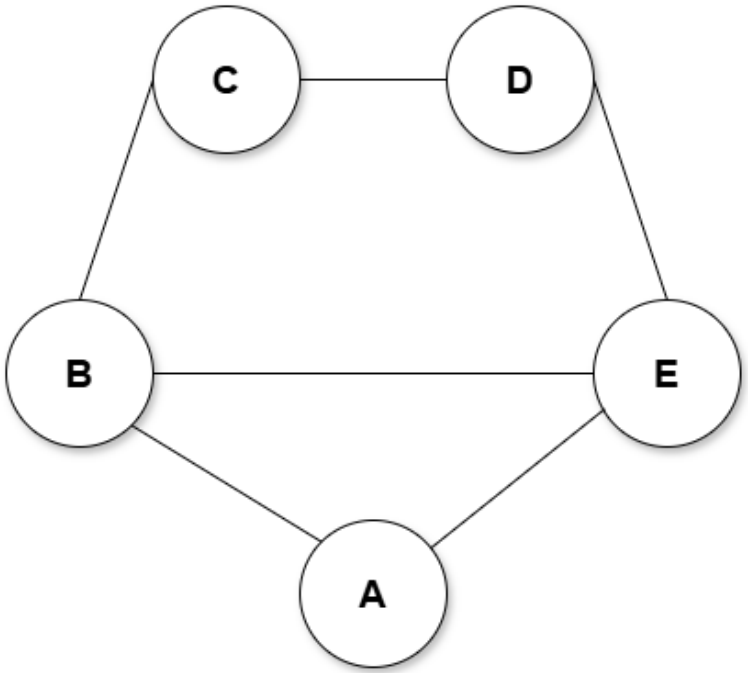
BFS

Cola:

Visitados:

A	B	C	D	E

Niveles = 0



BFS

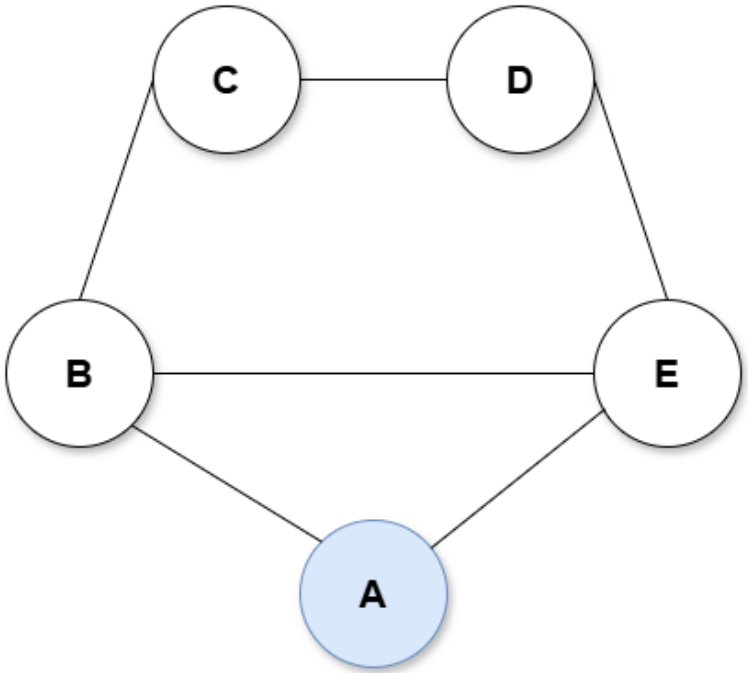
Cola:

A	None			

Visitados:

A	B	C	D	E

Niveles = 0



BFS

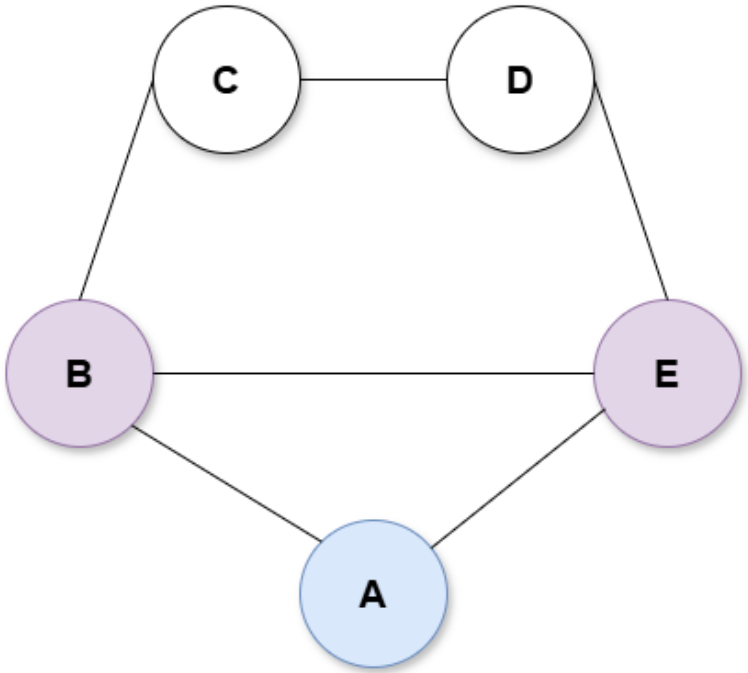
Cola:

A	None	B	E	

Visitados:

A	B	C	D	E

Niveles = 0



BFS

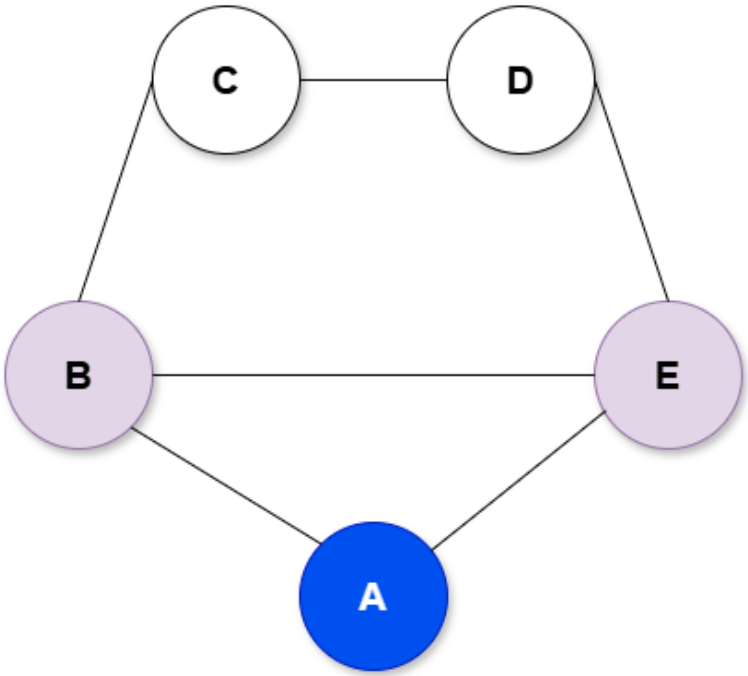
Cola:

None	B	E	None	

Visitados:

A	B	C	D	E

Niveles = 1



BFS

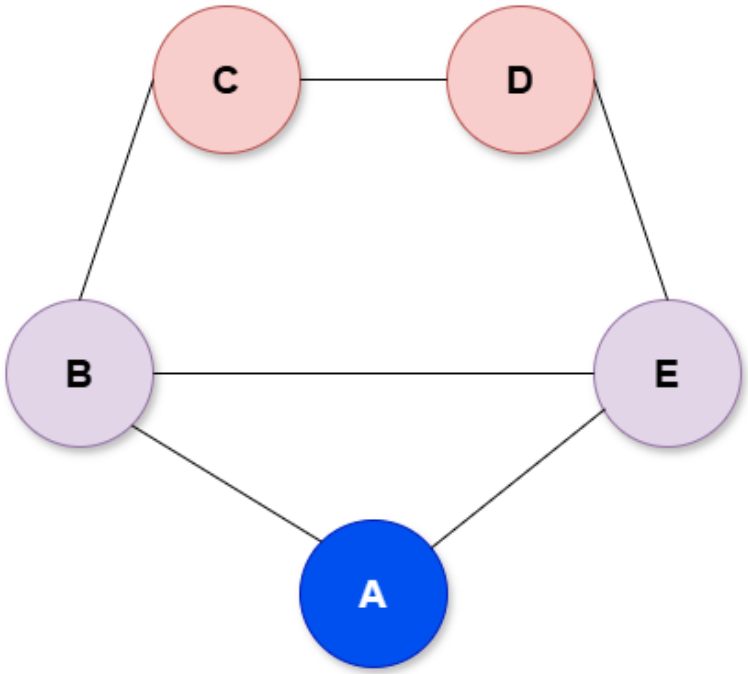
Cola:

B	E	None	C	D

Visitados:

A	B	C	D	E

Niveles = 1



BFS

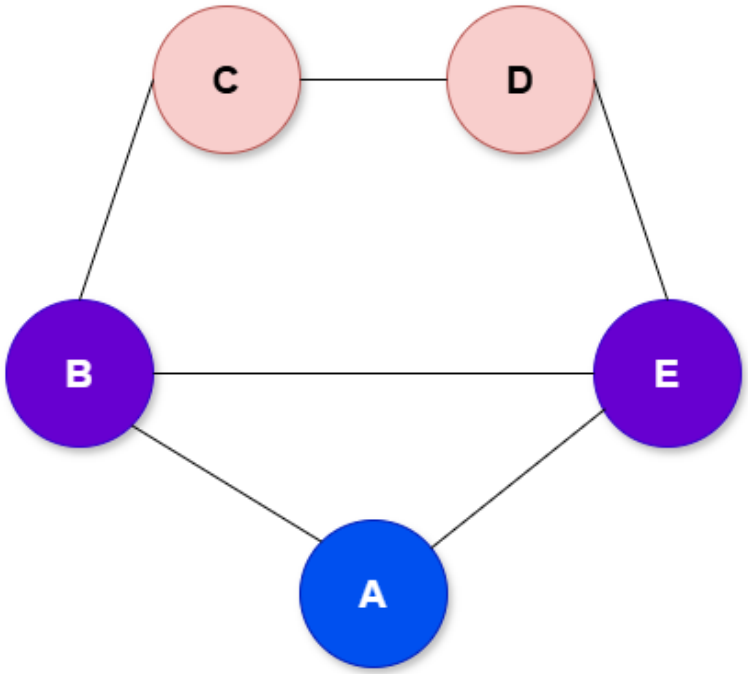
Cola:

None	C	D	None	

Visitados:

A	B	C	D	E

Niveles = 2



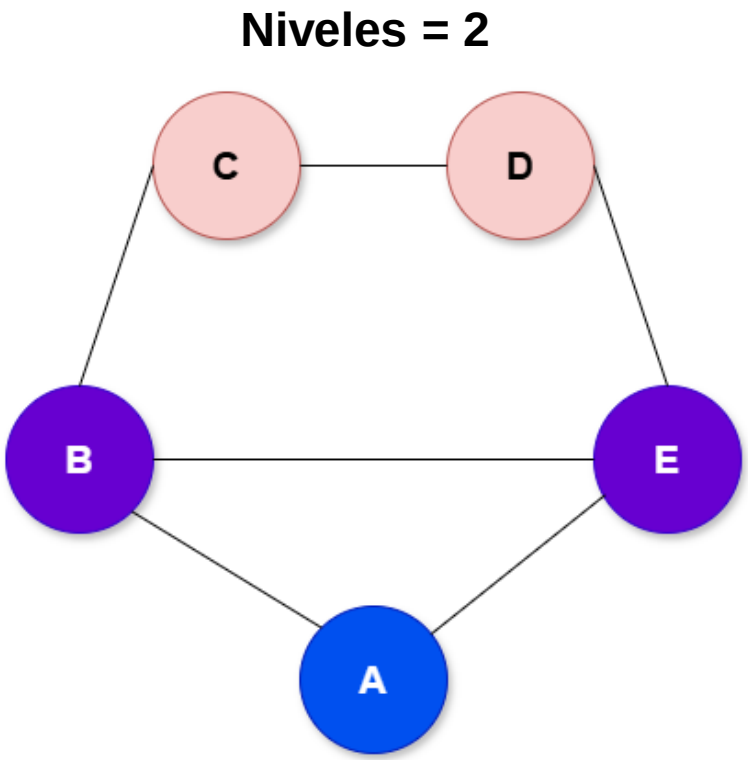
BFS

Cola:

C	D	None		

Visitados:

A	B	C	D	E



BFS

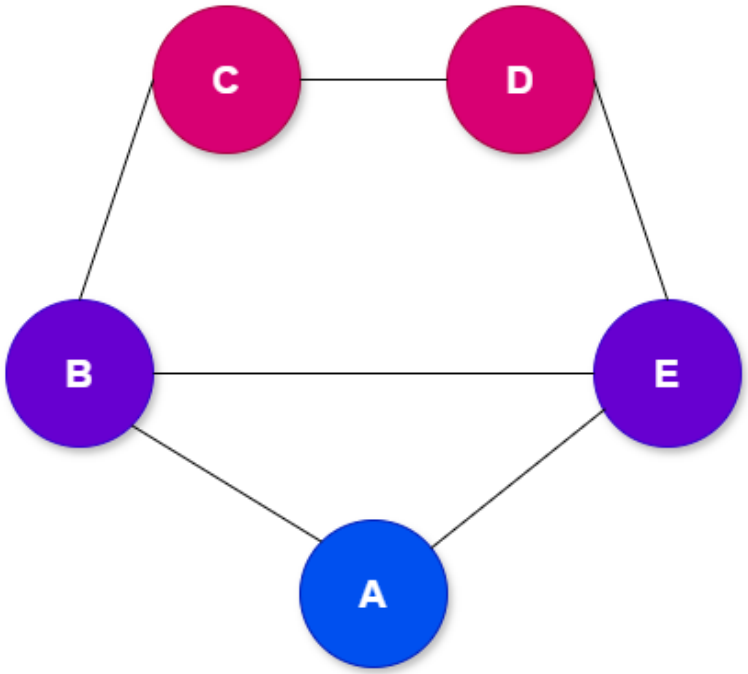
Cola:

None				

Visitados:

A	B	C	D	E

Niveles = 3



DFS

- Recorrido en profundidad: visitar toda una rama antes de retroceder

- Implementado con una pila y un array o set de visitados

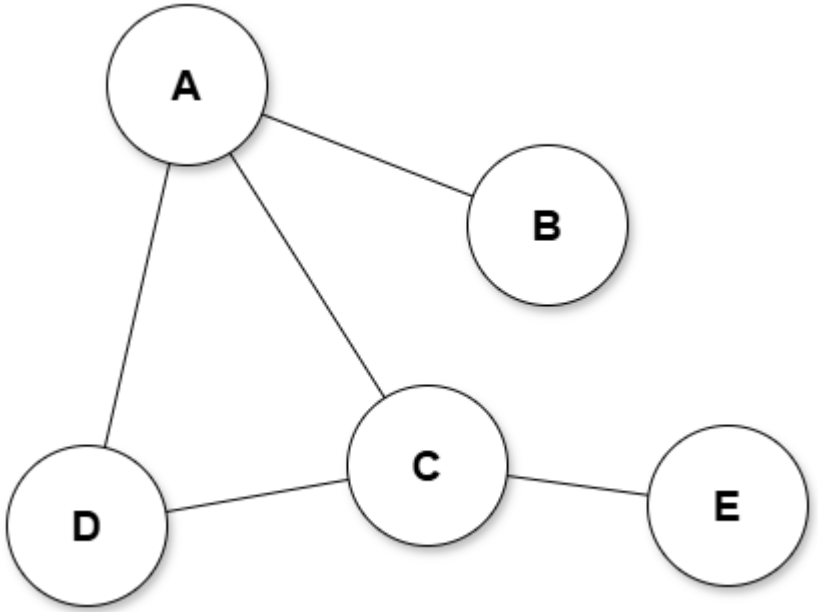
```
def dfs(grafo, visitados,origen):
    pila=deque()
    pila.append(origen)
    while pila:
        aux=pila.pop()
        if aux not in visitados:
            visitados.add(aux)
        for adj in reversed(grafo[aux]):
            if adj not in visitados:
                pila.append()
```

DFS

Pila:

Visitados:

A	B	C	D	E



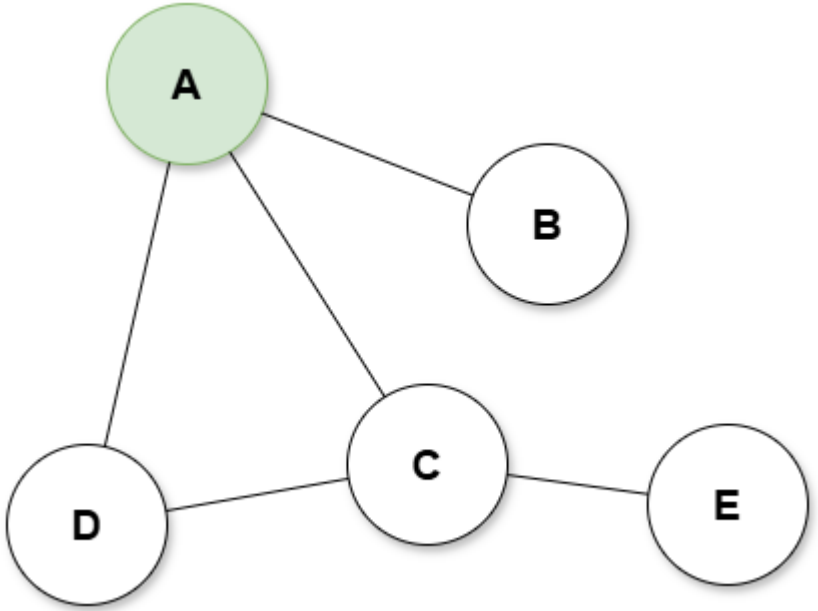
DFS

Pila:

A				

Visitados:

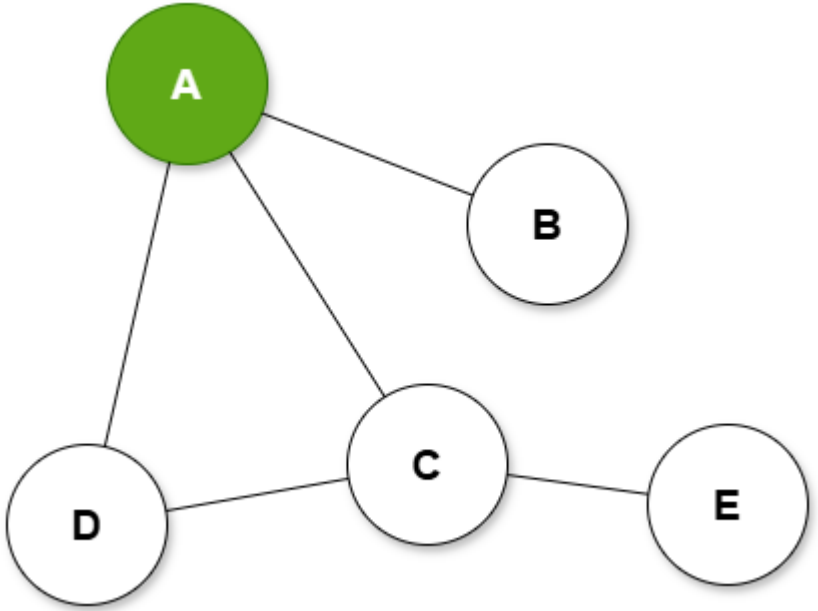
A	B	C	D	E



DFS

Pila:

Visitados:



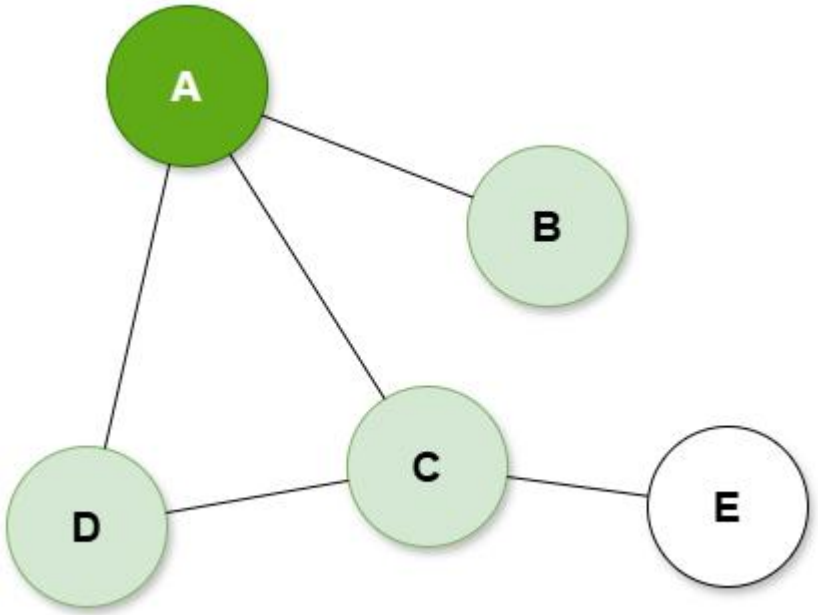
DFS

Pila:

D	C	B		

Visitados:

A	B	C	D	E



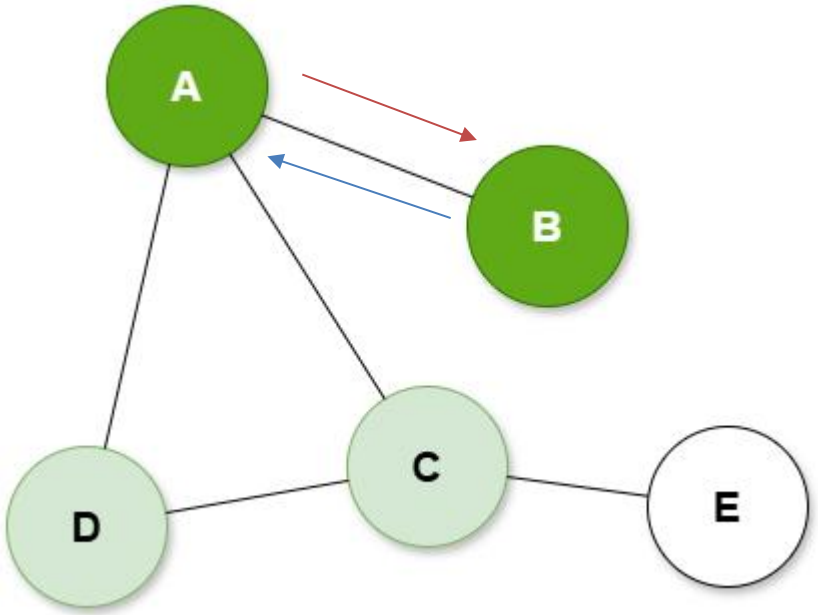
DFS

Pila:

D	C	B		

Visitados:

A	B	C	D	E



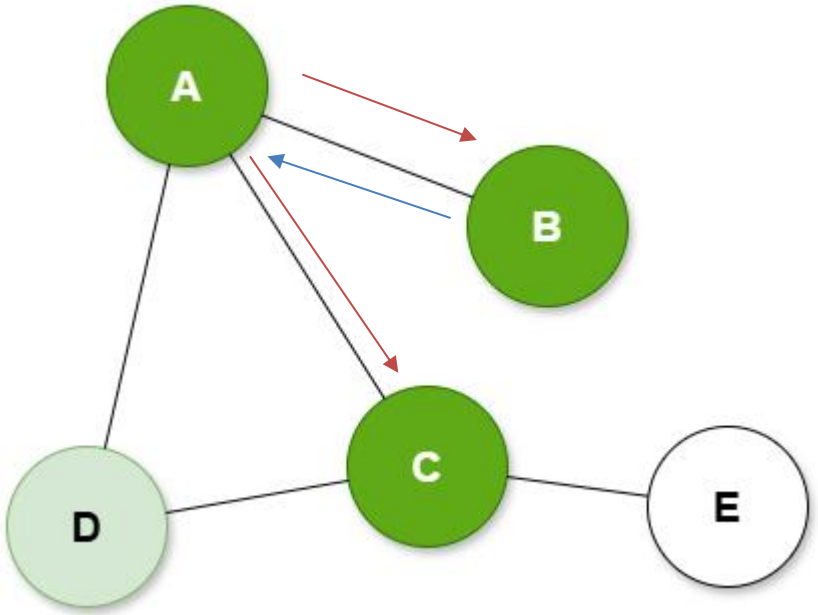
DFS

Pila:

D	C			

Visitados:

A	B	C	D	E



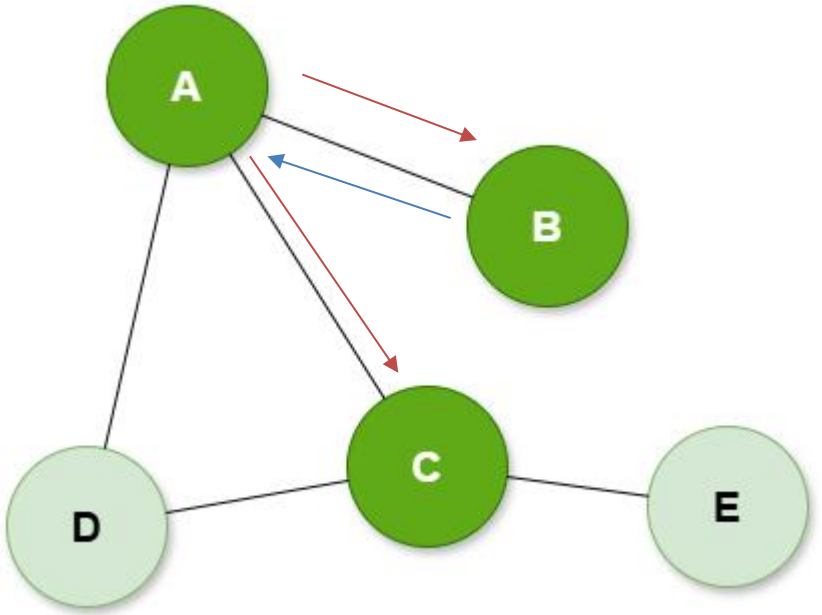
DFS

Pila:

D	E			

Visitados:

A	B	C	D	E



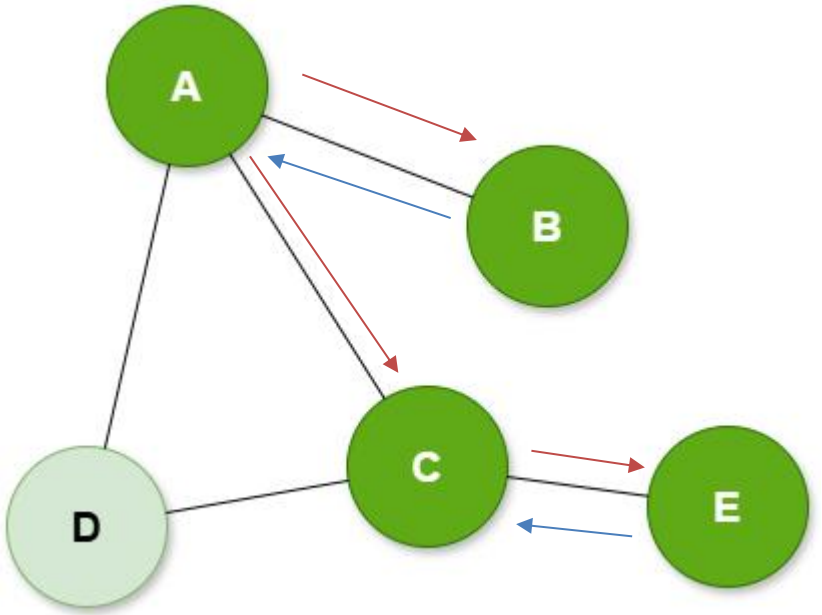
DFS

Pila:

D	E			

Visitados:

A	B	C	D	E



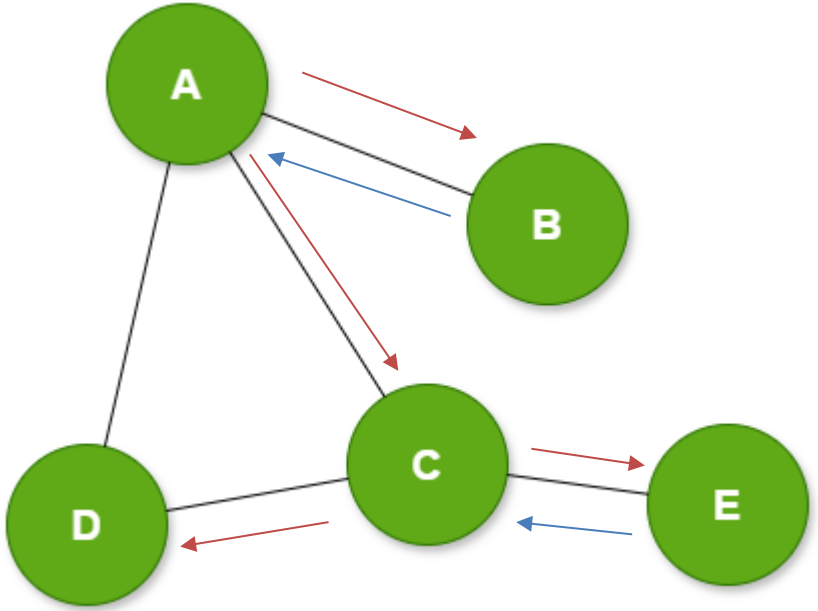
DFS

Pila:

D				

Visitados:

A	B	C	D	E



Componentes conexas

¿Cuántas componentes tiene el grafo?

¿Es un grafo conexo o fuertemente conexo?

Componentes conexas

¿Cuántas componentes tiene el grafo?

¿Es un grafo conexo o fuertemente conexo?

BFS o DFS

Caminos más cortos

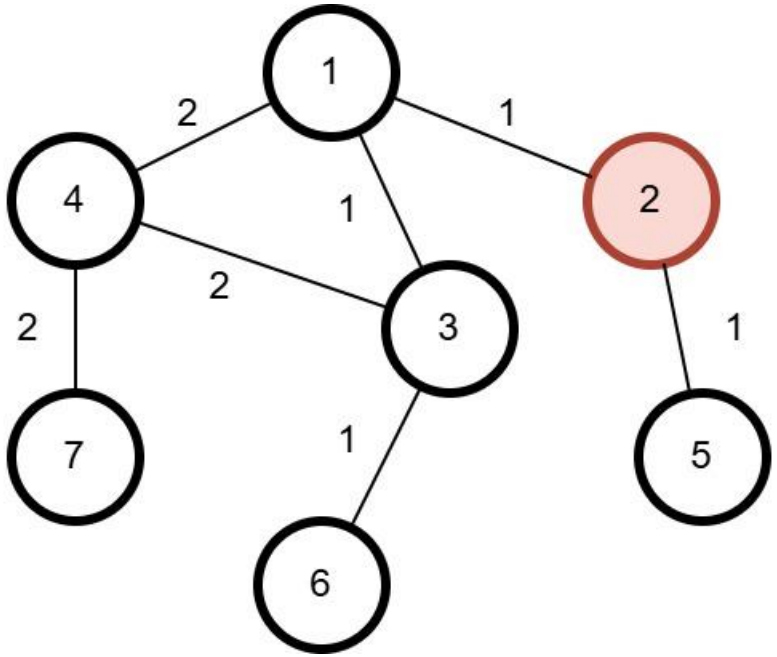
Problema:

Ha empezado una invasión de aliens y están estableciendo bases por todo el país. Los únicos lugares seguros son los que están suficientemente lejos de las bases de los aliens.

¿Cómo averiguamos cuáles son?

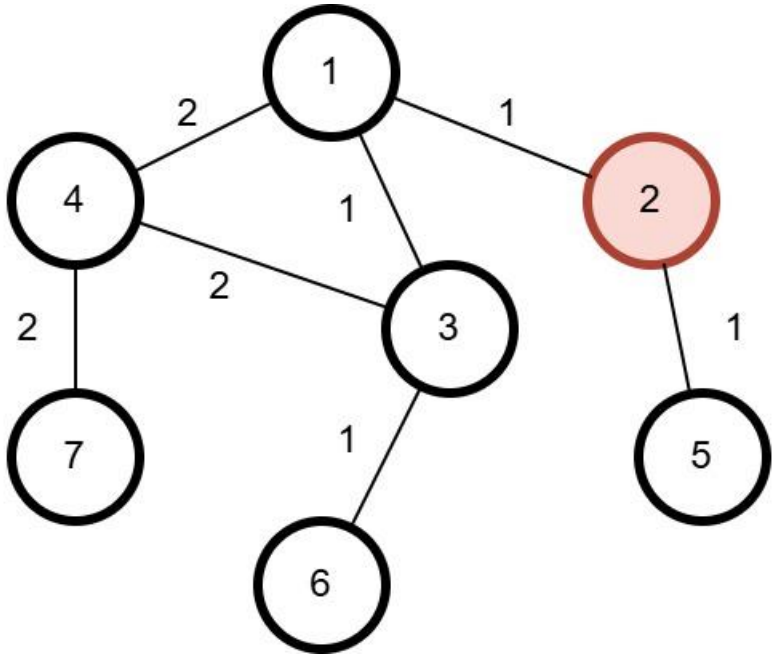
Caminos más cortos

Tenemos el mapa de ciudades con las distancias entre ellas, la ubicación de las bases de los aliens y la distancia segura, en este caso → **5**



Caminos más cortos

Para averiguar qué ciudades son seguras, tenemos que calcular la distancia más corta a cada una de las ciudades y ver si es mayor o igual a la distancia segura.



Caminos más cortos

¿Cómo calculamos las distancias?

ALGORITMO DE DIJKSTRA

Algoritmo voraz: si siempre escogemos el camino más corto, cuando lleguemos al final habremos llegado por el camino más corto

Dijkstra

Complejidad???

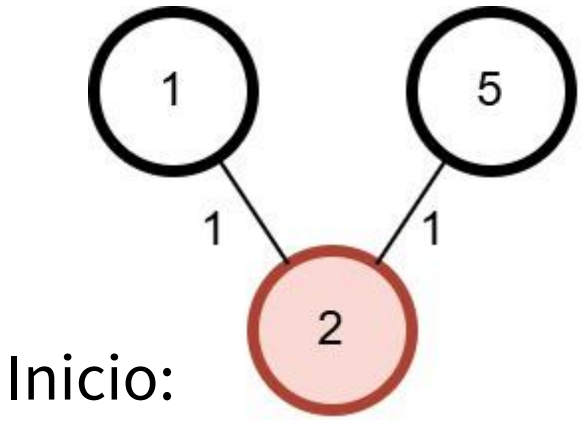
Si iteramos sobre todas las aristas en cada paso es muy costoso
→ $O(N^2)$

Cola de prioridad: guardamos los posibles nuevos caminos con su distancia

- Insertar y eliminar en PQ es O(log n)
- Por cada iteración **O(n log n)**

Dijkstra

1	∞
2	0
3	∞
4	∞
5	∞
6	∞
7	∞

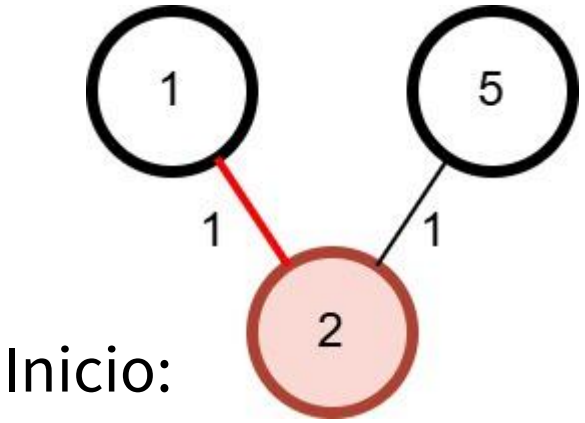


Inicio:

PQ	1,1	5,1

Dijkstra

1	1
2	0
3	∞
4	∞
5	1
6	∞
7	∞

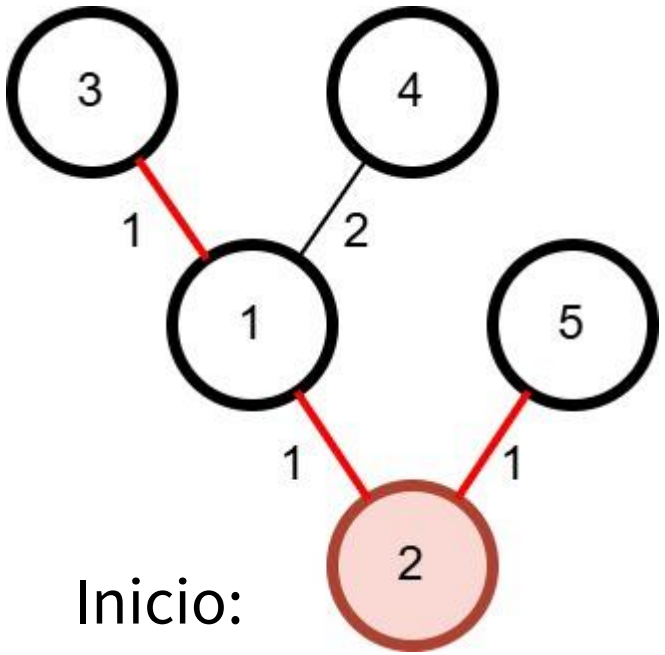


Inicio:

PQ	5,1

Dijkstra

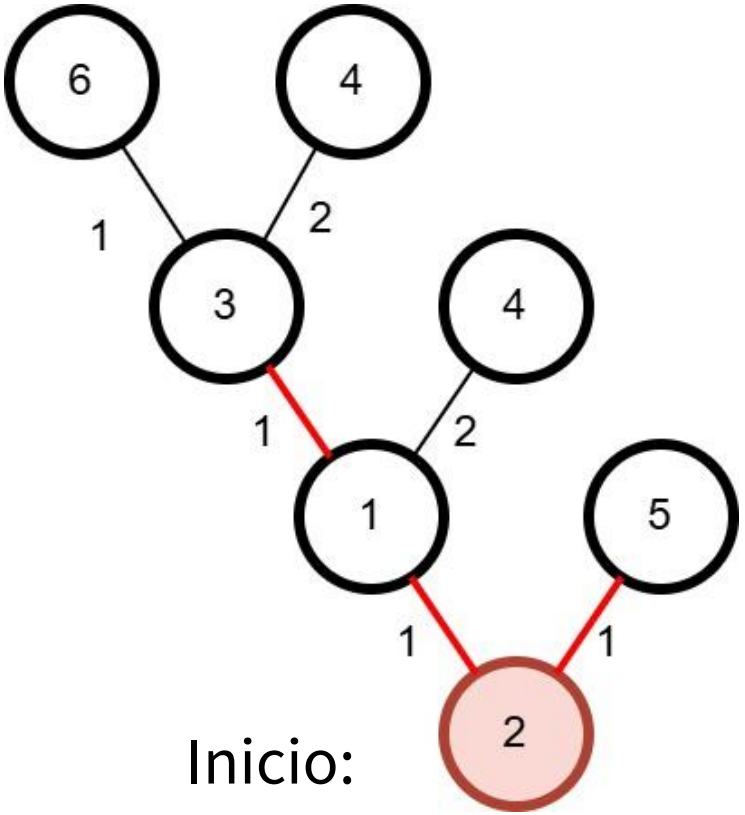
1	1
2	0
3	2
4	3
5	1
6	∞
7	∞



PQ	5,1	3,2	4,3

Dijkstra

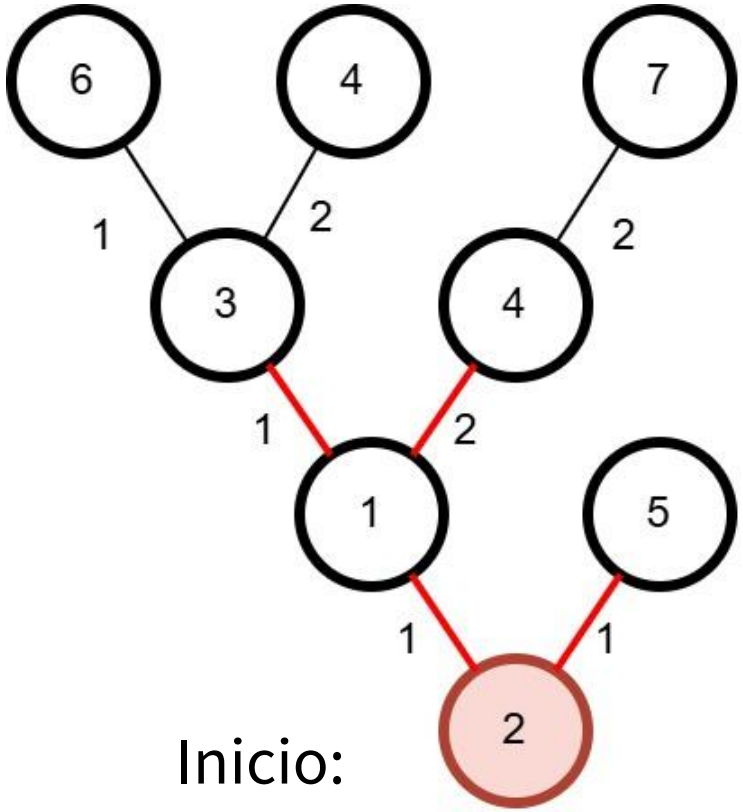
1	1
2	0
3	2
4	3
5	1
6	3
7	∞



PQ	3,2	4,3	6,3

Dijkstra

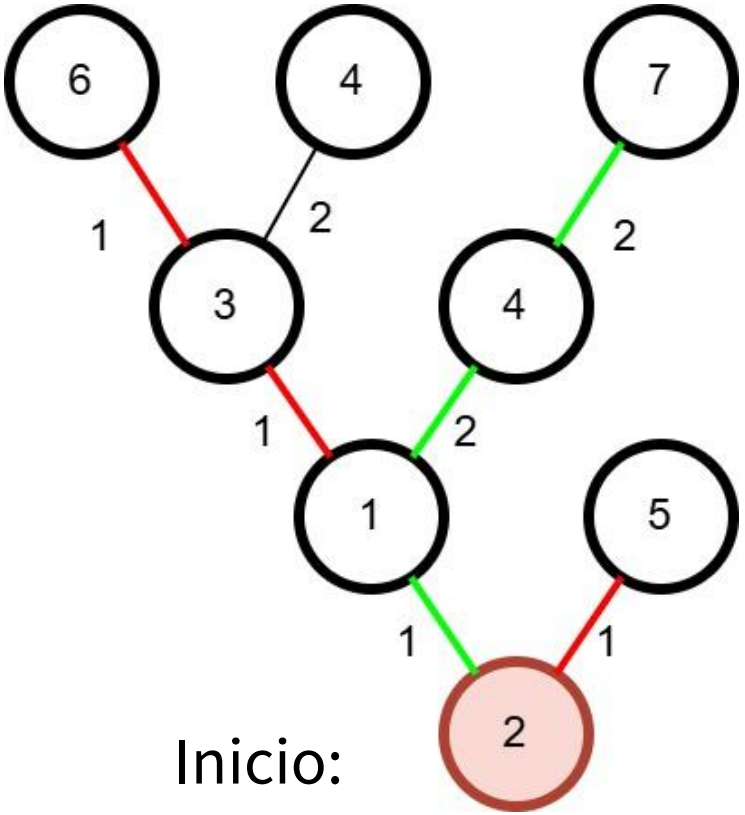
1	1
2	0
3	2
4	3
5	1
6	3
7	5



PQ	4,2	6,3	7,5

Dijkstra

1	1
2	0
3	2
4	3
5	1
6	3
7	5



PQ	

Algoritmo de Dijkstra

```
def dijkstra(graph, start):
    queue = []
    heapq.heappush(queue, (0, start))
    distances = [float('inf') for _ in range(len(graph))]
    distances[start] = 0

    while queue:
        current_distance, current_node = heapq.heappop(queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))
    return distances
```

Dijkstra

Problema propuesto:
https://open.kattis.com/problems/invasion

Encontrar el número de ciudades seguras según los aliens van construyendo sus bases

- Primera base: ¿cuántas ciudades son seguras?
- Siguiente base: ¿cuántas ciudades siguen siendo seguras? ¿Las que lo eran en el paso anterior están a suficiente distancia de la nueva base?

¿Preguntas?

HASTA LA SEMANA QUE VIENE!

@URJC_CP

@Dijkstraidos