presentación.js

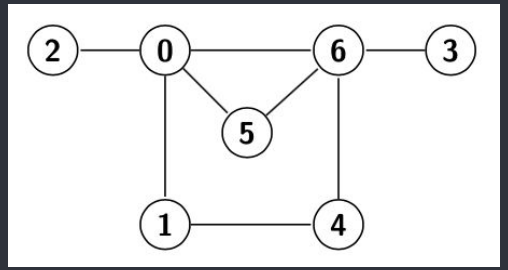
workshop.java

```
Club de Algoritmia US {
  [Universidad de Sevilla]

     Sesión 13: Grafos

}
```

Grafo

Es una estructura matemática empleada para modelar relaciones entre objetos. Consiste en:

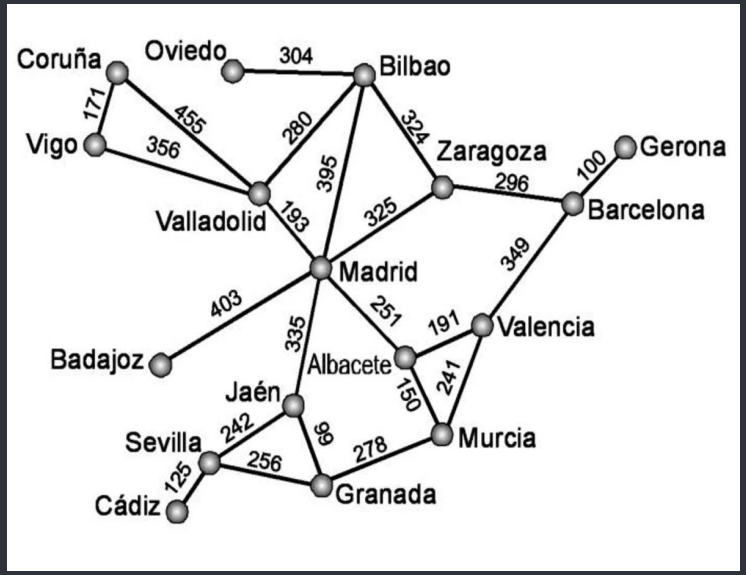
- **Vértices (o nodos):** Representan los objetos
- **Aristas (o arcos):** Representan las relaciones entre los objetos. Pueden ser dirigidas o no dirigidas

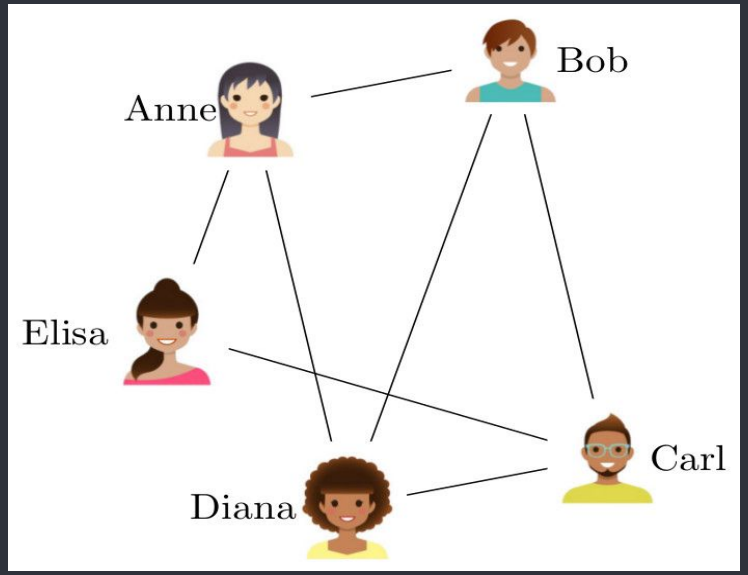


graph.py

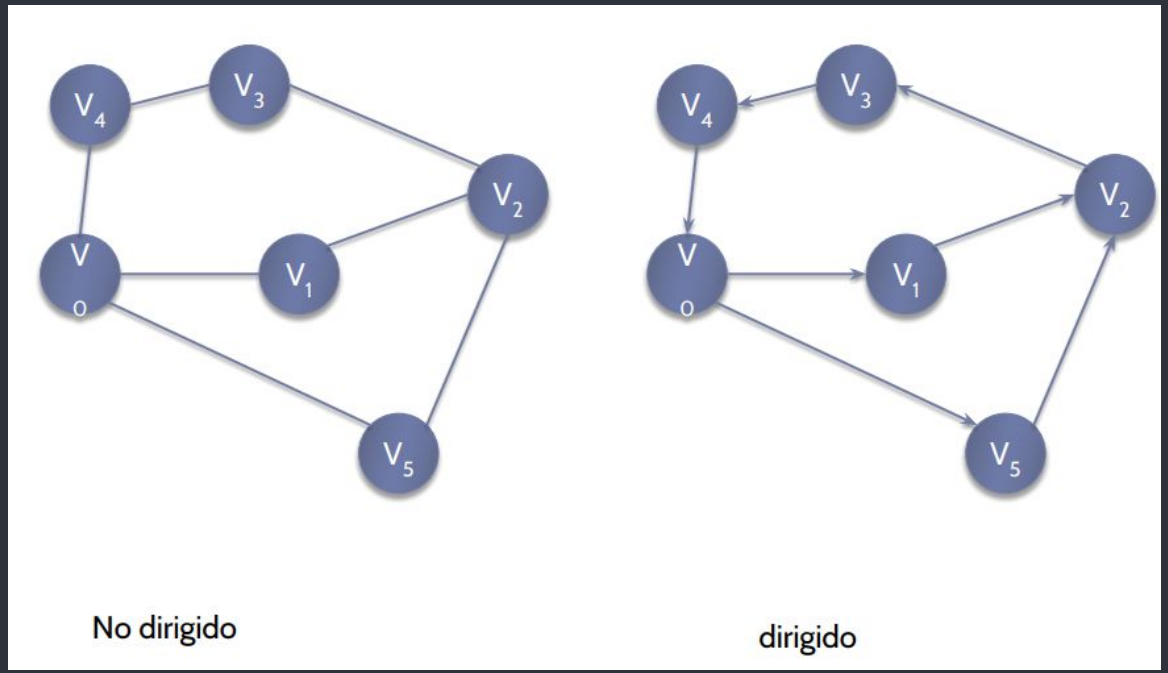
workshop.java

Grafos: ‘Ejemplos’



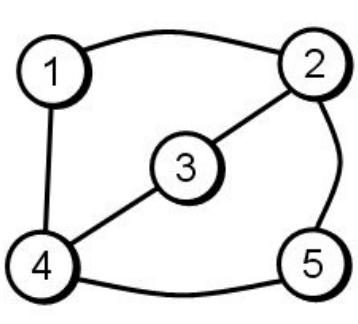


Grafo No Dirigido vs Dirigido



Grafos: ‘Implementación’

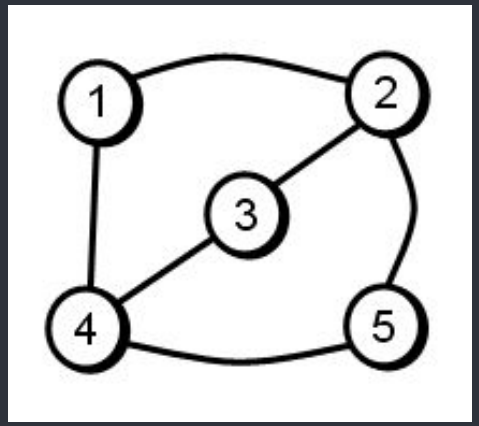
- **Matriz de adyacencias**: Se utiliza una matriz de tamaño n×n donde el elemento (i,j) contiene la distancia entre los vértices i y j del grafo.



M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Grafos: ‘Implementación’

- **Lista de adyacencias**: Por cada nodo se almacena una lista de los vértices adyacentes a i.



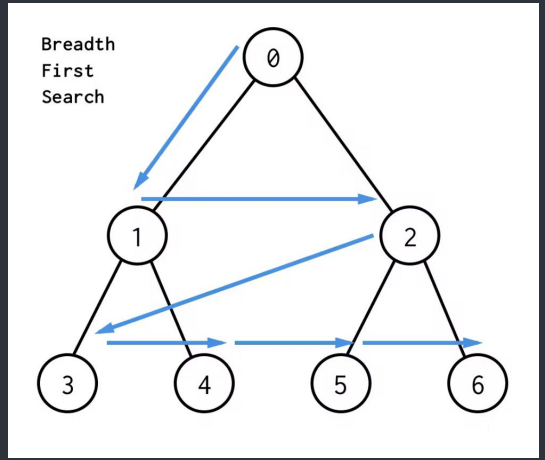
1	2, 4
2	1, 5
3	2, 4
4	1, 3, 5
5	2, 4

graph.py

workshop.java

Grafos: ‘Breadth-first search (BFS)’

Técnica para explorar todos los nodos de un grafo. Se emplea una cola para asegurar que los nodos se visiten en el orden de su distancia desde un nodo inicial.



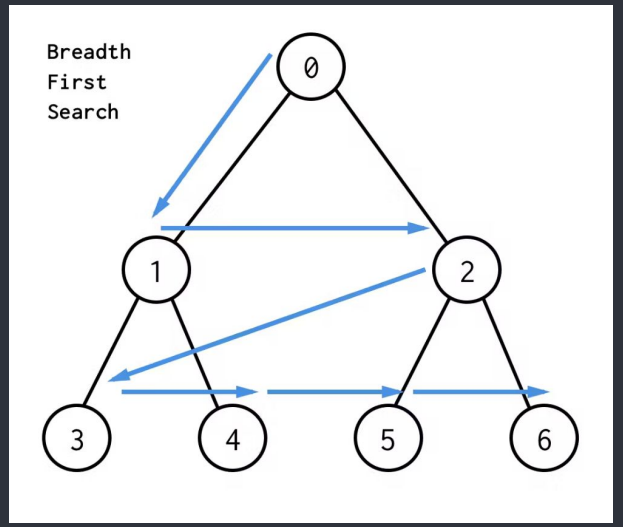
BFS: ‘Inicialización’

Escoge un nodo raíz x (puede ser cualquier nodo del grafo).

Crea un conjunto visitados para registrar los nodos visitados.

Agrega el nodo raíz x al conjunto visitados.

Crea una cola y encola el nodo raíz x.



graph.py

workshop.java

BFS: ‘Exploración’

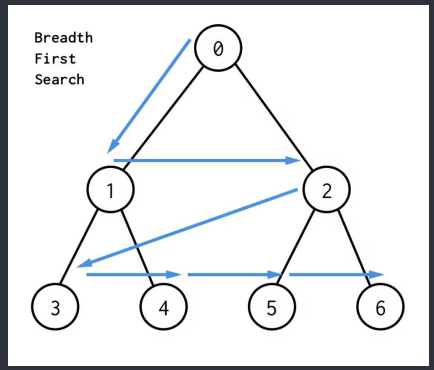
Mientras la cola no esté vacía:

  Extrae el primer nodo de la cola

  Para cada nodo vecino del nodo actual que no haya sido visitado:

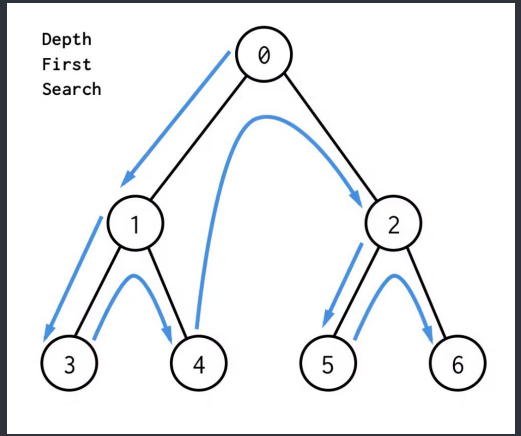
        Añádelo al conjunto visitados

        Agregar al final de la cola



Grafos: ‘Depth-first search (DFS)’

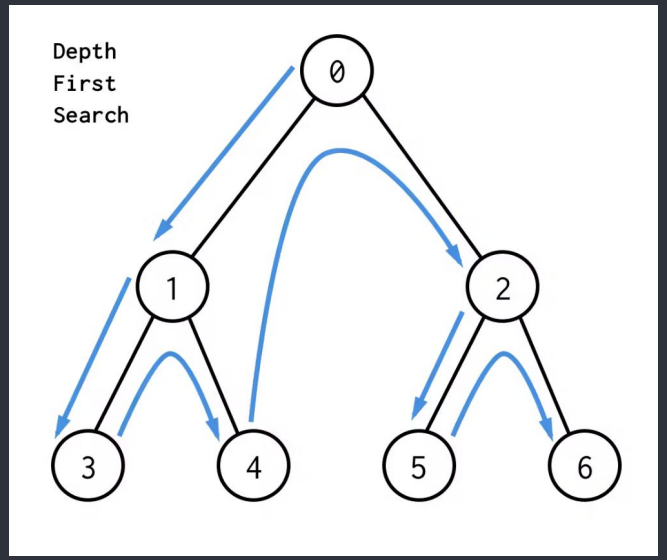
Técnica de exploración de grafos que explora las conexiones de un nodo de manera profunda hasta encontrarse en un camino sin salida, momento en el que vuelve a subir y explora el siguiente nodo no visitado.



DFS: ‘Inicialización’

Escoge un nodo raíz x (puede ser cualquier nodo del grafo)

Marca x como visitado



graph.py

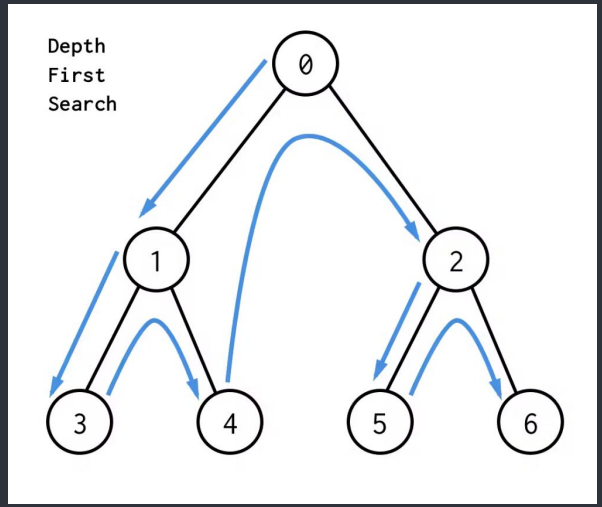
workshop.java

DFS: ‘Exploración’

Para cada nodo vecino no visitado v de x:

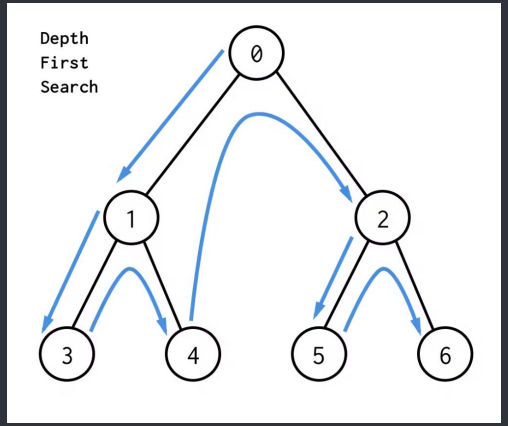
    Llamó recursivamente a DFS con

    v como nuevo nodo raíz



DFS: ‘Backtracking’

Una vez que se hayan explorado todas las ramas desde un nodo, regresa al nodo anterior y continúa la exploración desde ahí si es posible, hasta que todas las ramas hayan sido exploradas



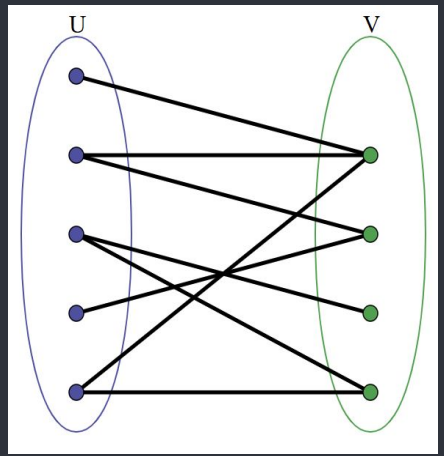
Visualización: ‘DFS y BFS’

Para visualizar cómo funcionan estas técnicas de exploración, podéis acceder al siguiente enlace:

https://visualgo.net/en/dfsbfs

Grafos: ‘Grafos Bipartitos’

Es un tipo de grafo cuyos vértices se pueden dividir en dos conjuntos distintos U y V, de manera que todas las aristas conectan un vértice en U con un vértice en V



Componentes conexas

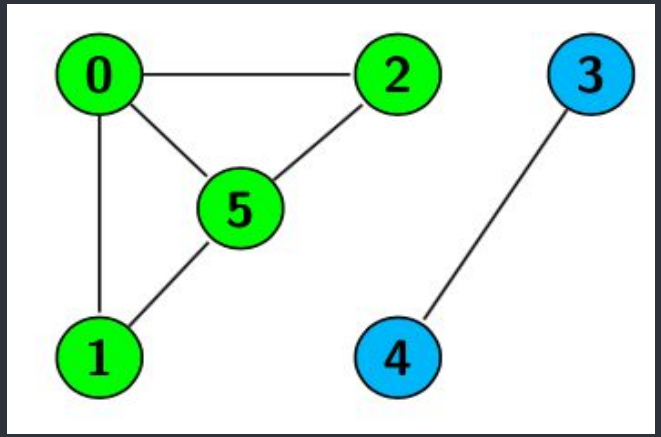
Las funciones de Búsqueda en Anchura (BFS) y Búsqueda en Profundidad (DFS) asumen que se puede llegar a todos los nodos del grafo comenzando desde la raíz elegida. Sin embargo, esto no tiene por qué ser siempre el caso.

graph.py

workshop.java

Componentes conexas

En un grafo no dirigido, una componente conexa es un grupo de nodos conectados entre sí. Esta componente contiene todas las conexiones de los nodos que la componen.

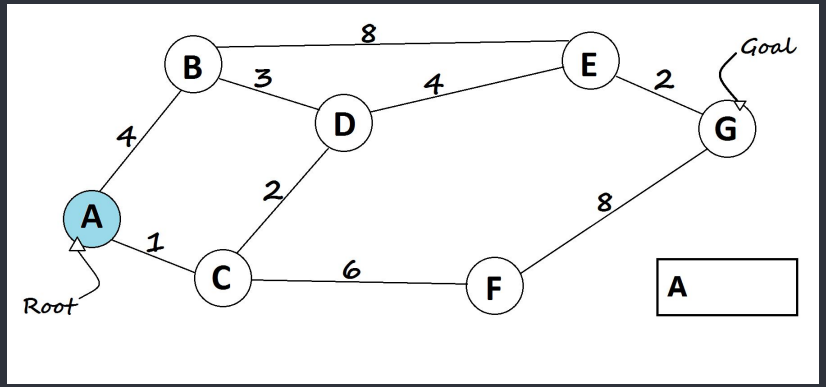


graph.py

workshop.java

Grafos: ‘Algoritmo de Dijkstra’

Algoritmo que sirve para encontrar, partiendo desde un nodo de salida u, la distancia más corta entre u y el resto de nodos de un grafo que tiene pesos en cada arista.

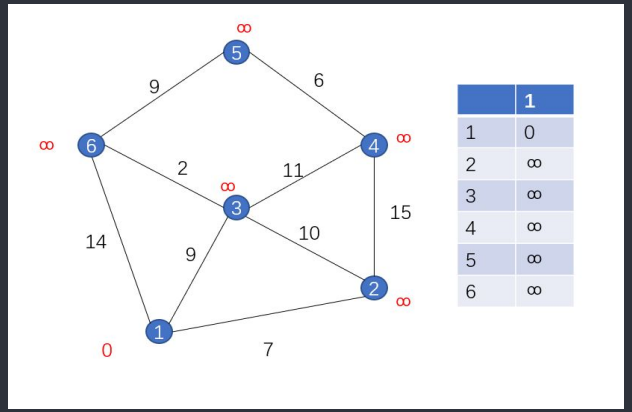


graph.py

workshop.java

Dijkstra: ‘Inicialización’

Se asigna una distancia infinita a todos los nodos excepto al nodo de inicio, al cual se le asigna una distancia de 0. Además, se inicializa un conjunto vacío de nodos visitados.



	1
1	0
2	∞
3	∞
4	∞
5	∞
6	∞

Dijkstra: ‘Bucle principal’

Se selecciona el nodo visitado con la distancia más pequeña

Marca este nodo como visitado

Para cada vecino no visitado del nodo seleccionado:

    Calcula la distancia provisional sumando la distancia del nodo seleccionado al peso del borde que conecta ambos nodos

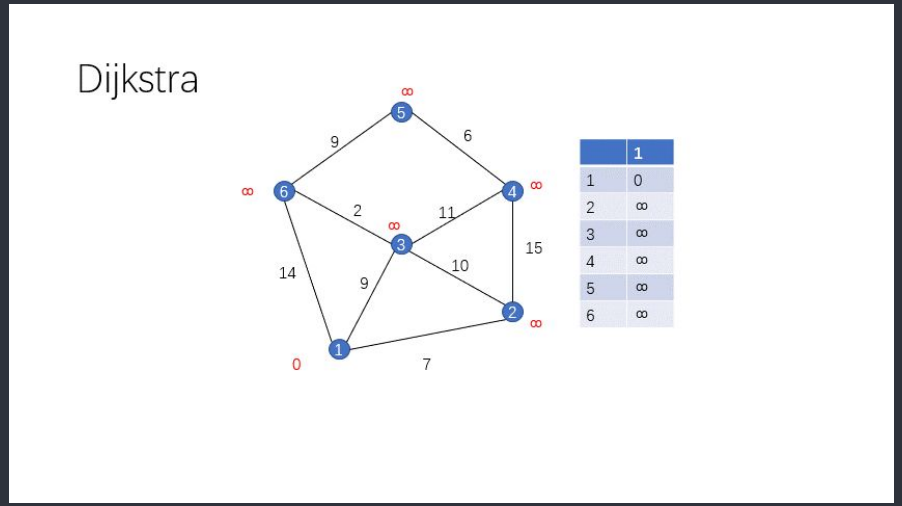
    Si la distancia provisional es menor que la distancia actual almacenada para el vecino, se actualiza la distancia del vecino con la distancia provisional

graph.py

workshop.java

Dijkstra: ‘Repetir’

Se repite el paso del bucle principal hasta que se visiten todos los nodos

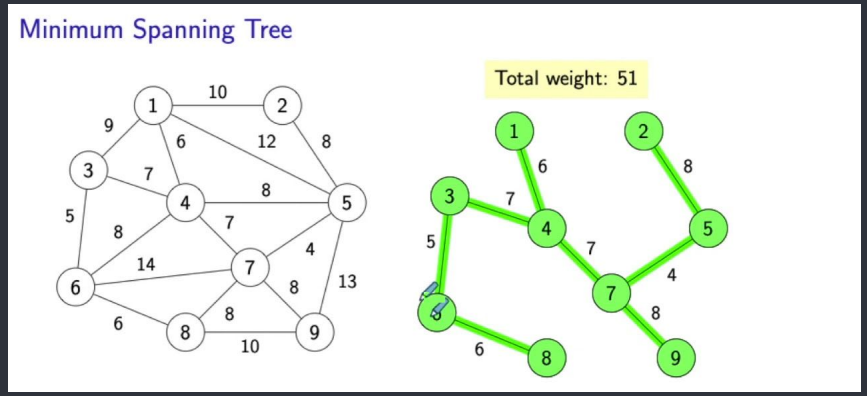


graph.py

workshop.java

Árbol de recubrimiento mínimo (MST)

Un árbol de expansión mínimo de un grafo conexo y no dirigido es un subconjunto de sus aristas que forman un árbol que incluye a todos los vértices y cuya suma de pesos es mínima.



Grafos: ‘Algoritmo de Kruskal

Ordena todas las aristas en orden no decreciente de acuerdo a sus pesos.

Inicializa un árbol vacío.

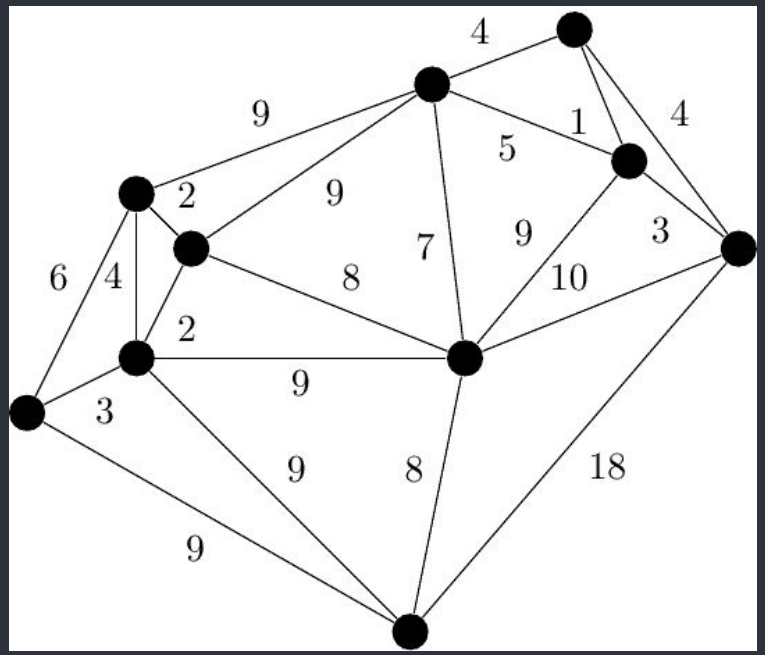
Recorre las aristas ordenadas y agrega cada arista al árbol si no crea un ciclo.

Repite hasta que todas las aristas hayan sido consideradas o hasta que el árbol contenga todos los vértices.

graph.py

workshop.java

Grafos: ‘Algoritmo de Kruskal

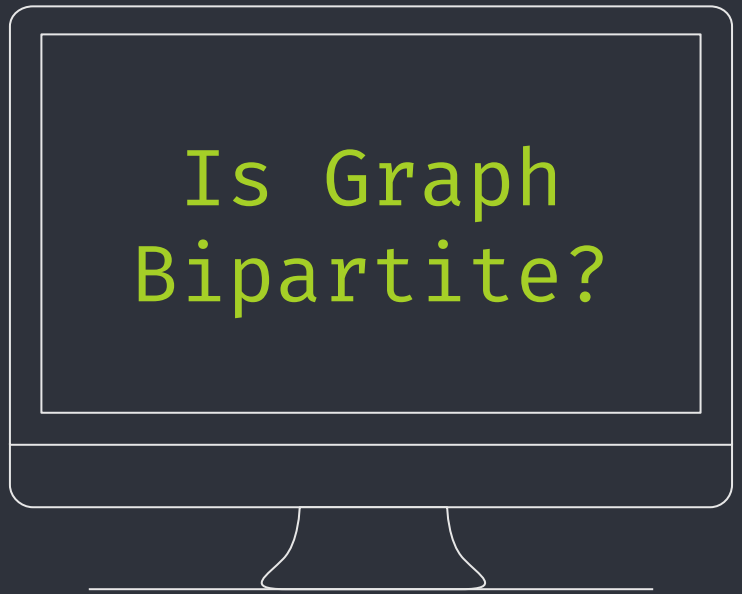


LeetCode - Número 547

Solución

```python
class Solution:

    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        n_nodes = len(isConnected)
        visited = set()
        provinces = 0
        while len(visited) < n_nodes:
            for i in range(n_nodes):
                if i not in visited:
                    boundary = {i}
                    break
            while len(boundary) > 0:
                i = boundary.pop()
                neighbours = {
                    j for j in range(n_nodes)
                    if isConnected[i][j]
                }
                boundary |= neighbours - visited
                visited.add(i)
            provinces += 1
        return provinces
```

LeetCode - Número 785



Solución

```python
def isBipartite(self, graph: List[List[int]]) -> bool:
    n_nodes = len(graph)
    colors = {}
    while len(colors) < n_nodes:
        for i in range(n_nodes):
            if i not in colors:
                start_node = i
                break
        if not self.connected_is_bipartite(graph, start_node, False, colors):
            return False
    return True
```

```python
def connected_is_bipartite(self, graph, node, color, colors) -> bool:
    if node in colors:
        if color != colors[node]:
            return False
        else:
            return True

    colors[node] = color
    for neighbour in graph[node]:
        if not self.connected_is_bipartite(graph, neighbour, not color, colors):
            return False

    return True
```

LeetCode - Número 1926

problemas.py

workshop.java

Prob: ‘Nearest Exit from Entrance in Maze’

{

Dado un laberinto y una posición inicial, encontrar la salida más próxima. Consideraremos salida a una celda que esté en el borde del laberinto que no sea pared y distinta de la inicial. Las celdas libres se representarán con “.” y las paredes con “+”

}

Prob: ‘Nearest Exit from Entrance in Maze’

{

Input: maze = [["+","+",".","+"],[".",".",".","+"],["+","+","+","."]
entrance = [1,2]

Output: 1

Input: maze = [["+","+","+"],[".",".","."],["+","+","+"]],
entrance = [1,0]

Output: 2

}

Solución

```python
class Solution:
    def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:
        rows, cols = len(maze), len(maze[0])
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        queue = deque([(entrance[0], entrance[1], 0)])
        visited = set((entrance[0], entrance[1]))

        while queue:
            row, col, steps = queue.popleft()

            if not (row == entrance[0] and col == entrance[1]) and (row == 0 or row == rows-1 or col == 0
or col == cols-1):
                return steps

            for dr, dc in directions:
                new_row, new_col = row + dr, col + dc

                if 0 <= new_row < rows and 0 <= new_col < cols and (new_row, new_col) not in visited and
maze[new_row][new_col] == '.':
                    queue.append((new_row, new_col, steps + 1))
                    visited.add((new_row, new_col))

        return -1
```

end.py

workshop.java

¡Gracias!

¡Nos vemos el curso que viene!

- Alguien se viene a escalar?