The Code >>>>>>

import queue

def addEdge(adj, u, v):

    adj[u].append(v)

    adj[v].append(u)

def BFSUtils(u, adj, visited):

    q = queue.Queue()

    visited[u] = True

    q.put(u)

    while not q.empty():

        u = q.get()

        print(u, end=' ')

        for i in range(len(adj[u])):

            if not visited[adj[u][i]]:

                visited[adj[u][i]] = True

                q.put(adj[u][i])

def bfs(adj, v):

    visited = [False] \* v

    for u in range(v):

        if not visited[u]:

            BFSUtils(u, adj, visited)

if \_\_name\_\_ == '\_\_main\_\_':

    v = 5

    adj = [[] for \_ in range(v)]

    addEdge(adj, 0, 4)

    addEdge(adj, 1, 2)

    addEdge(adj, 1, 3)

    addEdge(adj, 1, 4)

    addEdge(adj, 2, 3)

    addEdge(adj, 3, 4)

    print("BFS Traversal:")

    bfs(adj, v)

##### EXPLAINATION >>>>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Breadth-First Search (BFS) Explanation**

The given code is an implementation of **Breadth-First Search (BFS)** for an **undirected graph** using an adjacency list. BFS is a fundamental graph traversal algorithm that explores all neighbors of a node before moving to the next level.

**1. Algorithm: Breadth-First Search (BFS)**

**Steps of BFS**

1. **Initialize Data Structures**
   * Maintain a visited[] array to track visited nodes.
   * Use a **queue** (FIFO) to process nodes in order.
2. **Start BFS from an Unvisited Node**
   * Mark the node as **visited**.
   * Enqueue the node into the queue.
3. **Process the Queue Until Empty**
   * Dequeue a node.
   * Print (or process) the node.
   * Enqueue **all unvisited adjacent nodes** and mark them as visited.
4. **Repeat Until All Components Are Explored**
   * If the graph is disconnected, start BFS from any remaining unvisited node.

**2. How the Given Code Works**

**Functions Overview**

1. **addEdge(adj, u, v)**
   * Adds an **undirected edge** between u and v in the adjacency list.
2. **BFSUtils(u, adj, visited)**
   * Performs BFS starting from node u using a queue.
   * Ensures all connected nodes are visited.
3. **bfs(adj, v)**
   * Calls BFSUtils() for all unvisited nodes.
   * Ensures BFS runs on all **disconnected components**.

**3. Code Execution Breakdown**

**Graph Representation**

The adjacency list for the given graph:

0 --- 4

1 --- 2

| \ |

| \ |

| \ |

3 --- 4

Adjacency List:

0 → [4]

1 → [2, 3, 4]

2 → [1, 3]

3 → [1, 2, 4]

4 → [0, 1, 3]

**Step-by-Step Execution**

1. Start BFS from the lowest unvisited node (0).

vbnet

CopyEdit

Queue: [0]

Visited: [True, False, False, False, False]

Output: 0

1. Explore neighbors of 0 (4).

vbnet

CopyEdit

Queue: [4]

Visited: [True, False, False, False, True]

Output: 0 4

1. Explore neighbors of 4 (0, 1, 3).

vbnet

CopyEdit

Queue: [1, 3]

Visited: [True, True, False, True, True]

Output: 0 4 1 3

1. Explore neighbors of 1 (2).

vbnet

CopyEdit

Queue: [3, 2]

Visited: [True, True, True, True, True]

Output: 0 4 1 3 2

1. All nodes are visited, BFS completes.

**Final Output**

yaml

CopyEdit

BFS Traversal:

0 4 1 3 2

**4. Where BFS is Used in Real Life?**

BFS is widely used in various real-world applications:

**1. Shortest Path in an Unweighted Graph**

* BFS finds the shortest path in an **unweighted** graph (equal edge weights).
* Example: Finding the shortest route in a **road network**.

**2. Social Networking Websites**

* BFS helps find **connections** (e.g., finding mutual friends on Facebook).
* **Example:** Suggesting "People You May Know."

**3. Web Crawlers**

* **Search engines** use BFS to traverse the **internet graph**.
* **Example:** Google indexing webpages.

**4. AI & Chatbots**

* BFS is used in **game AI** to explore possible moves (e.g., **chess**).
* **Example:** AI solving puzzles like **Rubik’s Cube**.

**5. Network Broadcasting**

* BFS is used to **spread messages** across a network.
* **Example:** **Flood-fill algorithm** for **network packet routing**.

**6. Solving Maze Problems**

* BFS is used to find the **shortest escape path** in a **maze**.
* **Example:** **Google Maps shortest route calculation**.

**7. Recommendation Systems**

* **Example:** Netflix movie recommendations by finding **similar users**.

**5. Time & Space Complexity**

| **Operation** | **Complexity** |
| --- | --- |
| **Time Complexity** | O(V+E)O(V + E)O(V+E) (Each vertex & edge is processed once) |
| **Space Complexity** | O(V)O(V)O(V) (For queue & visited list) |

**6. Final Thoughts**

* **BFS is useful** for exploring graphs **level by level**.
* **It is best for** **finding shortest paths in unweighted graphs**.
* **Used in real-world applications** like **social networks, AI, and search engines**.

Now your BFS implementation is **correct, efficient, and useful** 🚀

Code of

class Graph:

    def \_\_init\_\_(self,vertices):

        self.adj = [[] for \_ in range(vertices)]

    def addEdge(self,s,d):

        self.adj[s].append(d)

        self.adj[d].append(s)

    def dfs\_rec(self,visited,s):

        visited[s] = True

        print(s,end = " ")

        for i in self.adj[s]:

            if not visited[i]:

                self.dfs\_rec(visited,i)

    def dfs(self):

        visited = [False] \* len(self.adj)

        for i in range(len(self.adj)):

            if not visited[i]:

                ####??? perform DFS from unvisited vertext

                self.dfs\_rec(visited,i)

if \_\_name\_\_ == "\_\_main\_\_":

    v = 6

    graph = Graph(v)

    edges = [(1,2), (2,0), (0,3), (4,5)]

    for edge in edges:

        graph.addEdge(edge[0], edge[1])

    print('complete DFS of the graph ')

    graph.dfs()

#### Explaintiaon

# ****Depth-First Search (DFS) Explanation****

The given code implements **Depth-First Search (DFS)** for an **undirected graph** using a **recursive approach**. DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

## ****1. Algorithm: Depth-First Search (DFS)****

### ****Steps of DFS****

1. **Initialize Data Structures**
   * Maintain a visited[] array to track visited nodes.
2. **Start DFS from an Unvisited Node**
   * Mark the node as **visited**.
   * Print (or process) the node.
   * Recur for **all unvisited adjacent nodes**.
3. **Repeat Until All Components Are Explored**
   * If the graph is **disconnected**, start DFS from any remaining unvisited node.

## ****2. How the Given Code Works****

### ****Class**** Graph ****Overview****

1. **\_\_init\_\_(self, vertices)**
   * Initializes an adjacency list for the graph.
   * self.adj = [[] for \_ in range(vertices)] creates an empty list for each vertex.
2. **addEdge(self, s, d)**
   * Adds an **undirected edge** between nodes s and d.
3. **dfs\_rec(self, visited, s)** (Recursive DFS)
   * Marks node s as **visited**.
   * Prints node s.
   * Recursively visits all unvisited adjacent nodes.
4. **dfs(self)** (DFS for All Components)
   * Initializes a visited[] array.
   * Calls dfs\_rec() for every unvisited node.
   * Ensures DFS is performed on **all disconnected components**.

## ****3. Code Execution Breakdown****

### ****Graph Representation****

Given edges:

(1,2), (2,0), (0,3), (4,5)

Graph structure:

1 --- 2

| |

0 --- 3

4 --- 5

Adjacency List:

0 → [2, 3]

1 → [2]

2 → [1, 0]

3 → [0]

4 → [5]

5 → [4]

### ****Step-by-Step Execution****

1. Start DFS from the lowest unvisited node (0).

Output: 0

Visited: [True, False, True, False, False, False]

1. Explore 0's neighbors (2, 3).

Output: 0 2 1

Visited: [True, True, True, False, False, False]

1. Backtrack, explore 3.

Output: 0 2 1 3

Visited: [True, True, True, True, False, False]

1. Move to the next unvisited node (4).

Output: 0 2 1 3 4

Visited: [True, True, True, True, True, False]

1. Explore 4's neighbor (5).

Output: 0 2 1 3 4 5

Visited: [True, True, True, True, True, True]

1. DFS is complete.

### ****Final Output****

complete DFS of the graph

0 2 1 3 4 5

## ****4. Where DFS is Used in Real Life?****

### ****1. Pathfinding in Mazes & Games****

* DFS explores all possible paths **before backtracking**.
* **Example:** AI solving a **maze** or **Sudoku solver**.

### ****2. Detecting Cycles in Graphs****

* DFS can detect **cycles** in a **directed** or **undirected** graph.
* **Example:** Detecting **deadlocks in operating systems**.

### ****3. Web Crawling (Search Engines)****

* **Example:** **Google Web Crawler** using DFS to explore links.

### ****4. Topological Sorting****

* DFS helps in **scheduling problems**.
* **Example:** **Course prerequisites** in **university schedules**.

### ****5. Finding Connected Components****

* DFS finds **disconnected clusters** in a graph.
* **Example:** **Finding social media communities**.

## ****5. Time & Space Complexity****

| **Operation** | **Complexity** |
| --- | --- |
| **Time Complexity** | O(V+E)O(V + E)O(V+E) (Each vertex & edge is processed once) |
| **Space Complexity** | O(V)O(V)O(V) (For recursion stack & visited list) |

## ****6. Final Thoughts****

* **DFS is useful** for **deep exploration** of graphs.
* **Best for** **pathfinding, detecting cycles, and solving puzzles**.
* **Used in real-world applications** like **AI, web crawling, and social networks**.

Your DFS implementation is **correct, efficient, and widely applicable** 🚀