"""

Assignment-A1: DFS & BFS

Problem Statement: Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

"""

from collections import deque

class Graph:

# Constructor to initalize an empty dictionary

def \_\_init\_\_(self):

self.graph = {}

# Add edges

def add\_edge(self, u, v):

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append(v)

self.graph[v].append(u) # Since the graph is undirected; Comment this line if graph is directed.

# Perform Depth First Search (DFS)

def dfs(self, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start, end=" ")

for neighbour in self.graph.get(start, []):

if neighbour not in visited:

self.dfs(neighbour, visited)

# Perform Breadth First Search (BFS)

def bfs(self, start):

visited = set()

visited.add(start)

queue = deque([start])

def bfs\_helper():

if not queue:

return

vertex = queue.popleft()

print(vertex, end=" ")

for neighbour in self.graph.get(vertex, []):

if neighbour not in visited:

visited.add(neighbour)

queue.append(neighbour)

bfs\_helper()

bfs\_helper()

def main():

g = Graph()

while True:

print("\n\n", "-"\*10, "MAIN MENU", "-"\*10)

print("1. Add edge")

print("2. Depth First Search (DFS)")

print("3. Breadth First Search (BFS)")

print("4. Exit")

choice = int(input("Choose an option (1-4):\t"))

print("-"\*32)

if choice == 1:

total = int(input("\nTotal edges to add:\t"))

for i in range(total):

print(f"EDGE {i+1} ->")

u = input(f"Enter first vertex for edge {i+1}: ")

v = input(f"Enter second vertex for edge {i+1}: ")

g.add\_edge(u, v)

print("\nVertices and its neighbours are:\t", g.graph)

elif choice == 2:

start = input("\nEnter starting vertex for DFS:\t")

print("Depth First Search (DFS):\t")

g.dfs(start)

elif choice == 3:

start = input("\nEnter starting vertex for BFS:\t")

print("Breadth First Search (BFS):\t")

g.bfs(start)

elif choice == 4:

print("\n## END OF CODE\n")

break

else:

print("\nPlease choose a valid option.")

main()

**✅ Code Explanation**

**1. Graph Class Initialization**

class Graph:

def \_\_init\_\_(self):

self.graph = {}

* A graph is represented as a **dictionary** where each key is a node and its value is a list of adjacent nodes (neighbors).
* This is an **adjacency list representation** of the graph.

**2. Adding Edges**

def add\_edge(self, u, v):

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append(v)

self.graph[v].append(u)

* Adds an **undirected edge** between u and v.
* Ensures each node has an entry in the dictionary.
* Adds each node to the other's adjacency list.

**3. Depth First Search (DFS) – Recursive**

def dfs(self, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start, end=" ")

for neighbour in self.graph.get(start, []):

if neighbour not in visited:

self.dfs(neighbour, visited)

* **Recursive implementation**.
* Uses a set to track visited nodes.
* Traverses as deep as possible before backtracking.
* This is a classic example of **recursive DFS**.

**4. Breadth First Search (BFS) – Recursive using Queue**

def bfs(self, start):

visited = set()

visited.add(start)

queue = deque([start])

def bfs\_helper():

if not queue:

return

vertex = queue.popleft()

print(vertex, end=" ")

for neighbour in self.graph.get(vertex, []):

if neighbour not in visited:

visited.add(neighbour)

queue.append(neighbour)

bfs\_helper()

bfs\_helper()

* Uses a **queue** (from collections.deque) to implement BFS.
* Visits nodes level by level (all neighbors first).
* BFS is usually **iterative**, but this version uses a **recursive helper function** to process the queue.

**5. Main Menu Function**

def main():

g = Graph()

while True:

...

* Provides a user menu to:
  + Add edges
  + Run DFS
  + Run BFS
  + Exit
* Takes user input and calls the appropriate function.

**🎯 Sample Output Flow**

1. Add edge

2. Depth First Search (DFS)

3. Breadth First Search (BFS)

4. Exit

Choose an option (1-4): 1

Total edges to add: 3

EDGE 1 ->

Enter first vertex: A

Enter second vertex: B

...

**🎤 Viva Questions with Easy Answers**

**1. What is DFS and how does it work?**

DFS (Depth First Search) is a graph traversal method that goes **deep into a path** before going to another path. It uses **recursion or a stack** and keeps track of visited nodes.

**2. What is BFS and how is it different from DFS?**

BFS (Breadth First Search) visits **all neighbors first** before going deeper.  
DFS goes **deep**, while BFS goes **wide**.  
BFS uses a **queue**, DFS uses **recursion or stack**.

**3. Why do we use a set for visited nodes?**

A set helps to **avoid visiting the same node again**. It also allows **fast checking** whether a node was visited.

**4. Why is BFS implemented using a queue?**

Because BFS visits nodes **in the order they were discovered** (first in, first out), a **queue** is the best structure for that.

**5. Can you explain the adjacency list representation of graphs?**

It's a dictionary where each node has a **list of neighbors**.  
Example: {'A': ['B', 'C'], 'B': ['A'], 'C': ['A']} means A connects to B and C.

**6. How is the graph stored in this code?**

As a **dictionary**. Each key is a node, and its value is a list of connected nodes (neighbors).

**7. How does recursion help in DFS?**

Recursion helps DFS to **go deep into a path automatically**, because each function call continues deeper until no more neighbors are left.

**8. What will happen if the graph has cycles?**

If cycles exist, DFS or BFS could get stuck in an **infinite loop**.  
That’s why we use a **visited set** to prevent visiting the same node again.

**9. What change would you make if the graph were directed?**

We would **remove** the line self.graph[v].append(u) in add\_edge(), so the edge goes **one way** only (from u to v).

**10. Why is BFS slower than DFS in recursive implementation?**

Recursive BFS is **not commonly used** because it can cause **stack overflow** for large graphs. It’s also **harder to write** than recursive DFS.

**11. What is the purpose of visited = None in DFS?**

It allows the function to **create a new visited set only once** when DFS starts.  
Otherwise, it would reset on every call.

**12. What is the use of deque in BFS?**

deque is a **double-ended queue**, and it is **faster** for adding/removing from both ends compared to a list.

**13. Why is self.graph.get(start, []) used instead of self.graph[start]?**

It **prevents errors** if the node doesn’t exist. get() returns an empty list if the node isn’t found.

**14. What is the output order of DFS vs BFS for a sample graph?**

For graph A - B - C, starting from A:

* **DFS**: A B C (goes deep)
* **BFS**: A B C (goes level by level – looks same here, but different for bigger graphs)

**15. What modifications would you make to track path or distance in BFS?**

Use a dictionary like distance[node] = distance[parent] + 1 and parent[node] = parent to keep track of paths and how far each node is.

Certainly! Here's the **time and space complexity** analysis for your **DFS and BFS implementation on an undirected graph**:

**✅ 1. Depth First Search (DFS)**

**🔸 Time Complexity:**

**O(V + E)**  
Where:

* **V** = number of vertices
* **E** = number of edges

🔹 **Why?**  
Each vertex is visited **once**, and each edge is considered **once** (in undirected graph).

**🔸 Space Complexity:**

**O(V)** for:

* Recursion stack (in worst case, depth could be equal to number of vertices),
* Visited set storing up to V nodes.

**✅ 2. Breadth First Search (BFS)**

**🔸 Time Complexity:**

**O(V + E)**

🔹 **Why?**

* Each node is **enqueued and dequeued** once, and
* All neighbors are checked, totaling E edge checks.

**🔸 Space Complexity:**

**O(V)** for:

* The queue (could hold up to V nodes),
* The visited set (up to V),
* Call stack (for your recursive BFS helper — note this is not typical for BFS, which is usually iterative).

**✅ Summary Table**

| **Traversal** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| DFS | O(V + E) | O(V) |
| BFS | O(V + E) | O(V) |

Based on the **SPPU (Savitribai Phule Pune University) 2019 Pattern**, here are common **viva questions** (with answers) that can be asked for the **DFS and BFS practical assignment**:

**🔹 General Graph Theory Questions**

1. **Q: What is a graph?**  
   **A:** A graph is a data structure that consists of a set of vertices (nodes) and a set of edges that connect pairs of vertices.
2. **Q: Difference between a directed and undirected graph?**  
   **A:** In a directed graph, edges have directions (A → B), while in an undirected graph, edges do not (A—B).
3. **Q: What is the difference between DFS and BFS?**  
   **A:** DFS explores as far as possible along one branch before backtracking (depth-wise), whereas BFS explores all neighbors at the present depth before moving to the next level (breadth-wise).
4. **Q: What are the real-life applications of DFS and BFS?**  
   **A:**
   * **DFS:** Solving puzzles (like mazes), cycle detection, topological sorting.
   * **BFS:** Shortest path in unweighted graphs, web crawling, peer-to-peer networks.

**🔹 Programming / Implementation Based Questions**

1. **Q: What data structure is used in DFS?**  
   **A:** DFS typically uses a **stack**, either explicitly or through **recursion**.
2. **Q: What data structure is used in BFS?**  
   **A:** BFS uses a **queue** to maintain the order of visiting nodes level by level.
3. **Q: What is the time and space complexity of DFS/BFS?**  
   **A:** Both have time complexity **O(V + E)** and space complexity **O(V)**.
4. **Q: What is the purpose of the visited set?**  
   **A:** It keeps track of nodes already visited to avoid infinite loops and repeated processing.

**🔹 Code-Specific Questions**

1. **Q: Why is the graph implemented using a dictionary?**  
   **A:** A dictionary (or hashmap) allows fast lookup and storage of adjacency lists for each vertex.
2. **Q: Can this code handle disconnected graphs?**  
   **A:** No, DFS or BFS starting from a single node will only cover the connected component. To handle disconnected graphs, you'd need to call DFS/BFS for every unvisited node.
3. **Q: What happens if the graph is directed?**  
   **A:** You should remove or comment the line self.graph[v].append(u) in the add\_edge() method to maintain edge direction.

**🔹 Advanced / Edge Cases**

1. **Q: What will happen if the start node is not present in the graph?**  
   **A:** The function may not output anything or may raise a KeyError (depending on implementation). It’s better to check if the start node exists.
2. **Q: How would you modify the code to track the path in BFS or DFS?**  
   **A:** You can maintain a dictionary (like parent) to keep track of each node's predecessor while traversing.

Would you like a **PDF of all these questions + answers** for offline revision?