Date : 6th January 2025

Experiment 1

AIM: WAP to implement DFS and BFS for traversing a graph from source node (S) to goal node (G), where source node and goal node is given by the user as an input.

Introduction:

Graph traversal is the process of visiting all the vertices or nodes of a graph in a systematic manner. The two most commonly used graph traversal techniques are **DFS (Depth First Search)** and **BFS (Breadth First Search)**:

1. **DFS (Depth First Search)**:
   DFS explores a graph by moving along one branch as far as possible before backtracking. It uses a **stack** (either explicit or the function call stack) to remember the vertices to visit.
   - Characteristics:
     - Visits nodes in depth-first manner.
     - Backtracks when no unvisited adjacent nodes remain.
     - Can be implemented recursively or iteratively.
2. **BFS (Breadth First Search)**:
   BFS explores a graph layer by layer, visiting all neighbors of a node before moving to their neighbors. It uses a **queue** to maintain the order of nodes to be visited.
   - Characteristics:
     - Visits nodes in breadth-first manner.
     - Ensures the shortest path (in terms of the number of edges) in an unweighted graph.
     - Always implemented iteratively using a queue.

DFS

Program:

```
"""required ds:
    graph: dictionary [key - node, value - neighbors]
    stack: to track nodes to visit
    visited: set to avoid repeated nodes
    path tracking to store the route
    traversal_path: list to store traversal order

    Firstly, make prog which dn detect graphs in undirected graphs:
"""

#functions for creating & manipulating a stack:
def create_stack():
    return []
```

```
def push(stack, item):
    stack.append(item)
    return stack

def pop(stack):
    if not is_empty(stack):
        return stack.pop(len(stack) - 1)
    return None

def is_empty(stack):
    return len(stack) == 0

def peek(stack):
    """View top item without removing it"""
    if not is_empty(stack):
        return stack[-1]
    return None


def dfs(graph, S, G):
    """This func. finds a path from a source node (S) to a goal node (G) if it
exists.
        Returns: a list containing the path bw S and G and also the traversal
map.
    """
    stack = create_stack()
    push(stack, (S, [S]))
    visited = set()
    traversal_path = []

    while stack:
        current, path = stack.pop()

        if current not in visited:
            traversal_path.append(current)
            visited.add(current)

            if current == G:
                print(f"Traversal order: {traversal_path}")
                return path

            for neighbor in graph[current]:
                if neighbor == G:
                    traversal_path.append(neighbor)
                    print(f"Traversal order: {traversal_path}")
                    return path + [neighbor]

                if neighbor not in visited:
```

```
                    new_path = path + [neighbor]
                    push(stack, (neighbor, new_path))

    print(f"Traversal Path: {traversal_path}")
    return "No path found."

def main():
    graph = {
        0: [1, 2, 3],
        1: [0],
        2: [0, 4],
        3: [0],
        4: [2]
    }

    S = int(input("Enter the source node: "))
    G = int(input("Enter the goal node: "))

    path = dfs(graph, S, G)
    print(f"Path from {S} to {G}: {path}")

main()
```

BFS

Program:

```
# Functions for creating & manipulating a queue:
def create_queue():
    return []

def enqueue(queue, item):
    queue.append(item)

def dequeue(queue):
    if not is_empty(queue):
        return queue.pop(0) # Pop the first element from the queue
    return None

def is_empty(queue):
    return len(queue) == 0

def bfs(graph, S, G):
    """This function finds the shortest path from a source node (S) to a goal
node (G) if it exists.
       Returns: a list containing the path between S and G and also the
traversal map.
    """
```

```
    queue = create_queue()
    enqueue(queue, (S, [S]))
    visited = set()
    traversal_path = []

    while queue:
        current, path = dequeue(queue)

        if current not in visited:
            traversal_path.append(current)
            visited.add(current)

            if current == G:
                print(f"Traversal order: {traversal_path}")
                return path

            for neighbor in graph[current]:
                if neighbor not in visited:
                    new_path = path + [neighbor]
                    enqueue(queue, (neighbor, new_path))

    print(f"Traversal Path: {traversal_path}")
    return "No path found."

def main():
    graph = {
        0: [1, 2, 3],
        1: [0, 5],
        2: [0, 4],
        3: [0],
        4: [2],
        5: [1]
    }

    S = int(input("Enter the source node: "))
    G = int(input("Enter the goal node: "))

    path = bfs(graph, S, G)
    print(f"Path from {S} to {G}: {path}")

main()
```

Applications:

DFS –

- Cycle Detection in Graphs

- Used in Directed Acyclic Graphs (DAGs) to determine the order of tasks (e.g., job scheduling, compiler design).
- **Path Finding in Mazes or Puzzles:** DFS explores all possible paths in problems like mazes, Sudoku solvers, and N-Queens.

BFS –

- **Social networking**: BFS can find people within a given distance of a person on social networking sites.
- **GPS**: BFS is used in GPS navigation systems to find neighboring locations
- **Graph theory**: BFS can be used to solve many problems in graph theory, such as finding the shortest path between two nodes.
- **Puzzle games**: BFS can be used to solve puzzle games like Rubik's Cubes.

Answer the following:

1. **If a question asks for a shortest path, or requires processing all nodes at a particular level / distance, you shall use** Breadth First Search**.**
2. **For problems other than the category mentioned in Q1, you must use** Depth First Search**. (1 point)**
3. **Enhance your code to detect cycle in undirected graph. (4 points)**

   DFS:

```
# Stack operations
def create_stack():
    return []

def push(stack, item):
    stack.append(item)
    return stack

def pop(stack):
    if not is_empty(stack):
        return stack.pop(len(stack) - 1)
    return None

def is_empty(stack):
    return len(stack) == 0

def peek(stack):
```

```
    """View top item without removing it"""
    if not is_empty(stack):
        return stack[-1]
    return None

def dfs_cycle(graph, node, visited, parent):
    """Helper function for cycle detection using DFS"""
    visited.add(node)

    for neighbor in graph[node]:
        # If neighbor not visited, explore it
        if neighbor not in visited:
            if dfs_cycle(graph, neighbor, visited, node):
                return True
        # If neighbor is visited and not parent, cycle exists
        elif neighbor != parent:
            print(f"Cycle detected! Back edge from {node} to
{neighbor}")
            return True
    return False

def detect_cycle(graph):
    """Detect if there's a cycle in an undirected graph"""
    visited = set()

    # Check each node to handle disconnected components
    for node in graph:
        if node not in visited:
            if dfs_cycle(graph, node, visited, None):
                return True
    return False

def dfs(graph, S, G):
    """Find path from source (S) to goal (G) if no cycles exist"""
    # First check for cycles
    if detect_cycle(graph):
        print("Graph contains cycles. Path finding aborted.")
        return None

    stack = create_stack()
    push(stack, (S, [S]))
    visited = set()
    traversal_path = []

    while stack:
        current, path = pop(stack)

        if current not in visited:
```

```
            traversal_path.append(current)
            visited.add(current)

            if current == G:
                print(f"Traversal order: {traversal_path}")
                return path

            for neighbor in graph[current]:
                if neighbor == G:
                    traversal_path.append(neighbor)
                    print(f"Traversal order: {traversal_path}")
                    return path + [neighbor]

                if neighbor not in visited:
                    push(stack, (neighbor, path + [neighbor]))

    print(f"Traversal order: {traversal_path}")
    return "No path found."

def main():
    # Example graph with a cycle (0-1-2-0)
    graph = {
        0: [1, 2],
        1: [0, 2, 3],
        2: [0, 1, 3],
        3: [1]
    }

    S = int(input("Enter the source node: "))
    G = int(input("Enter the goal node: "))

    path = dfs(graph, S, G)
    if path:
        print(f"Path from {S} to {G}: {path}")

if __name__ == "__main__":
    main()
```

BFS:

```
# Queue operations
def create_queue():
    return []

def enqueue(queue, item):
    queue.append(item)
```

```
def dequeue(queue):
    if not is_empty(queue):
        return queue.pop(0)
    return None

def is_empty(queue):
    return len(queue) == 0

def bfs_cycle(graph, node, visited, parent):
    """Helper function for cycle detection using BFS"""
    visited.add(node)
    queue = create_queue()
    enqueue(queue, (node, parent))

    while queue:
        current, parent_node = dequeue(queue)

        #check all the neighbors of the current node
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                enqueue(queue, (neighbor, current)) # Enqueue the
neighbor with current as its parent
            elif neighbor != parent_node:
                # If the neighbor is visited and it's not the parent, a
cycle is detected
                print("Cycle detected.")
                return True
    return False


def detect_cycle(graph):
    """Detect if there's a cycle in an undirected graph using BFS"""
    visited = set()

    for node in graph:
        if node not in visited:
            if bfs_cycle(graph, node, visited, None):
                return True
    return False

def bfs(graph, S, G):
    """Find shortest path from source (S) to goal (G) using BFS"""
    # First check for cycles
    if detect_cycle(graph):
        print("Graph contains cycles. Path finding aborted.")
        return None
```

```
    queue = create_queue()
    enqueue(queue, (S, [S]))
    visited = set()
    traversal_path = []

    while not is_empty(queue):
        current, path = dequeue(queue)

        if current not in visited:
            traversal_path.append(current)
            visited.add(current)

            if current == G:
                print(f"Traversal order: {traversal_path}")
                return path

            for neighbor in graph[current]:
                if neighbor not in visited:
                    enqueue(queue, (neighbor, path + [neighbor]))

    print(f"Traversal order: {traversal_path}")
    return "No path found."

def main():
    # Example graph with a cycle (0-1-3-2)
    graph = {
        0: [1, 2],
        1: [0, 2, 3],
        2: [0, 1, 3],
        3: [1]
    }

    S = int(input("Enter the source node: "))
    G = int(input("Enter the goal node: "))

    path = bfs(graph, S, G)
    if path:
        print(f"Path from {S} to {G}: {path}")

if __name__ == "__main__":
    main()
```

4. **Given two integers n and m, find all the stepping numbers in range [n,m] using DFS/BFS. (4 points)**

[A number is called stepping if the adjacent digits have an absolute difference of 1. For example, 321 is a stepping number, but 754 is not.]

- **Using DFS:**

```
def dfs(num, n, m, result):
    """
    Perform DFS to find stepping numbers starting with num
    """
    # If number is in range, add it to result
    if num >= n and num <= m:
        result.append(num)

    # If number exceeds m, stop exploration
    if num > m:
        return

    # Get last digit of current number
    last_digit = num % 10

    # Create next numbers by adding digits that differ by 1
    for next_digit in [last_digit - 1, last_digit + 1]:
        if next_digit >= 0 and next_digit <= 9:
            next_num = num * 10 + next_digit
            dfs(next_num, n, m, result)

def find_stepping_numbers(n, m):
    """
    Find all stepping numbers in range [n, m]
```

```
    Args:
        n (int): Lower bound
        m (int): Upper bound

    Returns:
        list: Sorted list of stepping numbers in the range
    """
    result = []

    # Handle 0 separately
    if n <= 0 <= m:
        result.append(0)

    # Start DFS with single digits (1-9)
    for i in range(1, 10):
        dfs(i, n, m, result)

    return sorted(result)

# Example usage
def main():
    n = int(input("Enter the lower bound: "))
    m = int(input("Enter the higher bound: "))
    numbers = find_stepping_numbers(n, m)
    print(f"Stepping numbers in range [{n}, {m}]:")
    print(numbers)

main()
```