Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT] THANOS

Experiment # 3: Find the path to reach the target from a source node in given Graph

Aim/Objective:

Implementing BFS for finding the path from a source to the goal node in the graph.

Description:

Students will create a simple graph and traverse the graph using Breadth-first search. BFS is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbor nodes. Then, it selects the nearest node and explores all the unexplored nodes.

Pre-Requisites:

a) Basic understanding of search algorithms.
b) Familiarity with Python programming language.
c) Knowledge of Graph traversing.

Pre-Lab:

1. What is Breadth-First Search (BFS)? Explain the basic idea behind BFS and how it explores a graph or a maze.

- BFS is a graph traversal algorithm that explores all neighbors of a node before moving to the next level. It explores the graph in layers, starting from the source node and visiting all nodes at distance **k** before moving to nodes at distance **k+1**. It ensures the shortest path in an unweighted graph.

2. What data structure(s) are commonly used in BFS? Describe their purpose and how they help in implementing the algorithm efficiently.

- **Queue**: BFS uses a queue to maintain the list of nodes to visit next. Nodes are dequeued, explored, and their unvisited neighbors are enqueued. This ensures the nodes are explored in the correct order, layer by layer.

3. How can a grid-based maze be represented in a data structure? Discuss the options and explain which one you would choose for this implementation.

- A grid-based maze can be represented using a **2D array** or **matrix**, where each cell is a node, and walls or paths are represented by values like 0 (empty) or 1 (blocked). This structure allows easy access to neighboring cells during traversal.

4. What are the steps involved in implementing BFS to solve a maze problem? Provide a high-level overview of the algorithm.

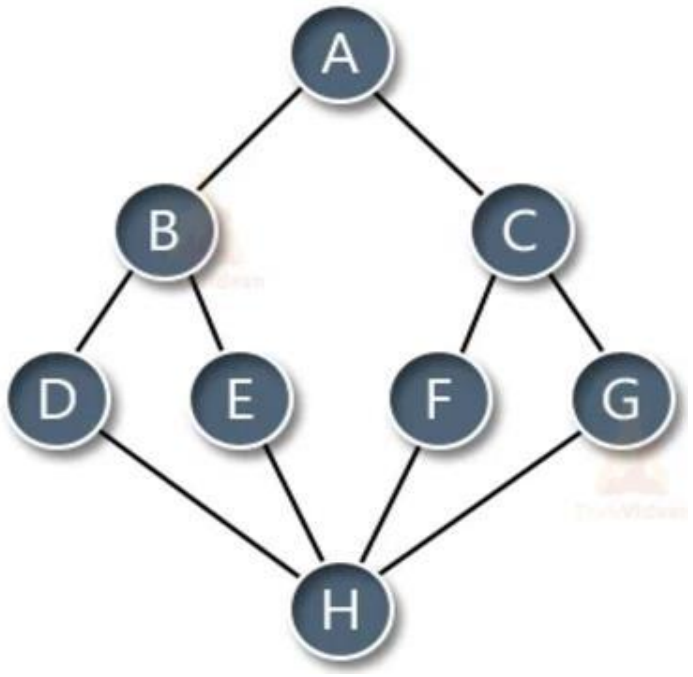
- Steps:
  1. Initialize a queue and enqueue the start position.
  2. Mark the start position as visited.
  3. While the queue is not empty:
     - Dequeue a cell, check if it's the goal.
     - Explore neighboring cells (up, down, left, right).
     - Enqueue unvisited neighboring cells and mark them as visited.
  4. If the goal is found, backtrack to get the solution path.

5. What is the purpose of marking visited cells in BFS? How can you keep track of visited cells efficiently during the traversal process?

- Marking visited cells prevents revisiting and getting stuck in loops. It ensures each cell is processed once. A **2D array** (same size as the maze) can be used to store visited information, marking cells as visited (e.g., setting the value to 1).

In-Lab:

Implement Depth-First Search using Python and find a path from *node 0* to *node 7* for the graph below using DFS.



Procedure/Program:

```
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F', 'G'],
   'D': ['H'],
   'E': ['H'],
   'F': ['H'],
   'G': ['H'],
   'H': []
}

def dfs(graph, start, goal, path):
   path.append(start)
   if start == goal:
      return True
   for neighbor in graph[start]:
      if neighbor not in path:
```

```
            if dfs(graph, neighbor, goal, path):
                return True
    path.pop()
    return False

start_node = 'A'
goal_node = 'H'
path = []

if dfs(graph, start_node, goal_node, path):
    print("Path from A to H:", path)
else:
    print("No path found from A to H.")
```

OUTPUT

```
Path from A to H: ['A', 'B', 'D', 'H']
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT] THANOS

- **Data and Results:**

Data

This program demonstrates Depth First Search (DFS) traversal implementation.

Result

The path from node A to H is successfully found.

- **Analysis and Inferences:**

Analysis

DFS explores graph nodes deeply before backtracking for unvisited paths.

Inferences

DFS efficiently identifies paths but may not guarantee shortest paths.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT] THANOS

Sample VIVA-VOCE Questions (In-Lab):

1. What are the different algorithms available for finding the shortest path in a graph, and how do they differ?

- **Dijkstra's, Bellman-Ford, Floyd-Warshall, A***. They differ in time complexity and handling negative weights.

2. Explain the working of Dijkstra's algorithm and discuss its time complexity.

- It finds the shortest path by expanding the nearest node. Time complexity is **O(E log V)**.

3. How does the Breadth-First Search (BFS) algorithm work for finding the shortest path in an unweighted graph?

- BFS explores nodes level by level, ensuring the shortest path. Time complexity is **O(V + E)**.

4. What are the advantages and limitations of using Depth-First Search (DFS) for pathfinding in a graph?

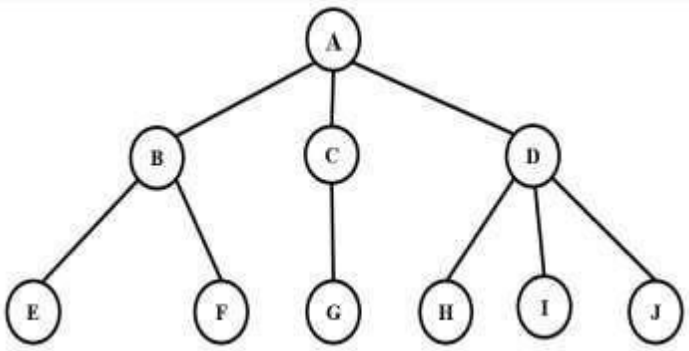
- **Advantages**: Simple and explores all paths. **Limitations**: Doesn't guarantee the shortest path.

5. Describe how you would implement the A search algorithm for pathfinding in a graph, and explain the role of the heuristic function. *

- A* combines Dijkstra's with a heuristic (**f(n) = g(n) + h(n)**) to guide the search, optimizing performance. The heuristic directs the search efficiently.

Post-Lab:

Implement BFS for the given tree.



- **Procedure/Program:**

```
from collections import deque

class Node:
    def __init__(self, value):
        self.value = value
        self.children = []

def bfs(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        current = queue.popleft()
        result.append(current.value)

        for child in current.children:
            queue.append(child)

    return result
```

```
a = Node("A")
b = Node("B")
c = Node("C")
d = Node("D")
e = Node("E")
f = Node("F")
g = Node("G")
h = Node("H")
i = Node("I")
j = Node("J")

a.children = [b, c, d]
b.children = [e, f]
c.children = [g]
d.children = [h, i, j]

print("BFS Traversal:", bfs(a))
```

OUTPUT

```
BFS Traversal: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT] THANOS

- **Data and Results:**

Data

This is a tree structure with nodes and connections defined.

Result

The BFS traversal visits nodes layer by layer sequentially.

- **Analysis and Inferences:**

Analysis

BFS explores all neighbor nodes before moving to deeper levels.

Inferences

Breadth-first traversal is useful for level-order processing in trees.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date