Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Spring, Year: 2025), B.Sc. in CSE (Day)

LAB REPORT NO 02

Course Title: Artificial IntelligenceLab

Course Code: CSE-316 Section: 222-D6

Student Details

	Name	ID
1.	MD FAHIM SARKER MRIDUL	221902369

Submission Date : 11-03-2025

Course Teacher's Name : Kazi Hasnayeen Emad

[For Teachers use only: Don't Write Anything inside this box]

Lab Report Status

Marks : ………… **Signature :.....................**

Comments:....................... **Date :.....................**

Title: BFS, DFS, and Shortest Path Finding using BFS

Objective: To understand and implement Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms and utilize BFS to find the shortest path in an unweighted graph.

Theory:

1. **Breadth-First Search (BFS):**
   - Explores all neighbors of a node before moving to the next level.
   - Uses a queue (FIFO) for traversal.
   - Efficient for finding the shortest path in an unweighted graph.
2. **Depth-First Search (DFS):**
   - Explores as far as possible along a branch before backtracking.
   - Uses a stack (LIFO) or recursion.
   - Useful for topological sorting and cycle detection.
3. **Shortest Path using BFS:**
   - BFS ensures the shortest path in an unweighted graph by visiting nodes in increasing order of distance.
   - Stores parent nodes to reconstruct the path.

Implementation:

- Implement BFS from a given graph.
- Implement DFS from a given graph.
- BFS is used to find the shortest path from a source to a destination.

Code of BFS:

```
from queue import Queue

def bfs(graph, start_node):
    visited = set()
    q = Queue()

    q.put(start_node)
    visited.add(start_node)

    while not q.empty():
        node = q.get()
        print(node, end=" ")
```

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

print("BFS Traversal Order:")
bfs(graph, 'A')
```

```
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                q.put(neighbor)
```

Output:

```
print("BFS Traversal Order:")
bfs(graph, 'A')
```

```
BFS Traversal Order:
A B C D E F G H
```

Code of DFS:

```
from collections import deque

def dfs(graph, start):
    stack = deque([start])  # Using deque as a stack (LIFO)
    visited = set()

    while stack:
        node = stack.pop()  # Pop from the right (LIFO)
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            stack.extend(reversed(graph[node]))
```

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

dfs(graph, 'A')
```

Output:

```
dfs(graph, 'A')
```

```
A B E F C G D H
```

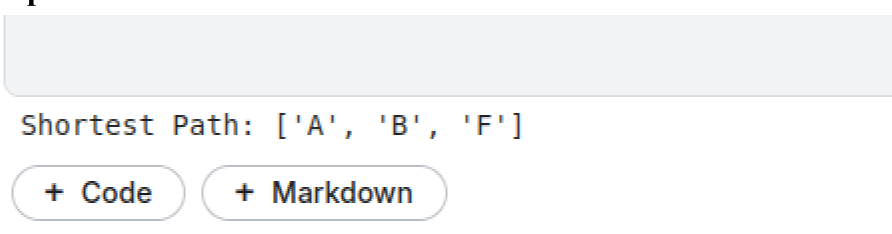
+ Code + Markdown

Code of Shortest path finding using BFS:

```
from collections import deque

def bfs_shortest_path(graph, start, goal):
    queue = deque([[start]])      visited = set()

    while queue:
        path = queue.popleft()        node =
path[-1]
        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                new_path = path + [neighbor]
                queue.append(new_path)
    return None
```

```
graph = {
    'A': {'B', 'C','D'},
    'B': {'E','F'},
    'C': {'G'},
    'D': {'H'},
    'E': {'B' },
    'F': { 'B'},
    'G': { 'C'},
    'H': {'D' }
}
shortest_path = bfs_shortest_path(graph, 'A',
'F')
print("Shortest Path:", shortest_path)
```

Output:



Conclusion:

BFS explores level-wise, ensuring the shortest path in an unweighted graph.DFS explores deeper first, useful for other applications like cycle detection.BFS-based shortest path finding is efficient and easy to implement in such graphs.