Experiment No. 3

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

Overview:

Breadth-First Search (BFS) is an algorithm used for traversing or searching tree or graph data structures. The key characteristic of BFS is that it explores all the neighbor nodes at the present depth level before moving on to the nodes at the next depth level. This means it systematically expands outward from the starting node and explores all of its neighbors at the current depth before moving deeper into the graph.

Algorithm Steps:

1. **Initialization:**
   - Start by enqueueing the initial or starting node into a queue.
   - Mark the starting node as visited.

2. **Traversal:**
   - While the queue is not empty, repeat the following steps:
     - Dequeue a node from the front of the queue. This node represents the current node being explored.
     - Visit and process the current node.
     - Enqueue all unvisited neighboring nodes of the current node into the queue.
     - Mark each of these neighboring nodes as visited.

3. **Termination:**
   - Terminate the algorithm when the queue becomes empty, indicating that all reachable nodes have been visited.

Key Points:

- BFS is optimal for finding the shortest path in an unweighted graph.

- It guarantees that the shortest path from the starting node to any other node is found first.

- BFS can be used to find the shortest path, level by level, from the starting node to any goal node.

2. **Uniform Cost Search (UCS):**

Overview:

Uniform Cost Search (UCS) is an algorithm used for traversing or searching tree or graph data structures where the edges have associated costs or weights. Unlike BFS, which explores nodes in a breadth-first manner, UCS selects the node with the lowest total cost from the starting node to that node. It ensures that the shortest path to a node is found before moving on to explore other nodes.

Algorithm Steps:

1. **Initialization:**

   - Start by enqueueing the initial or starting node into a priority queue ordered by path cost.

   - Initialize the cost of the starting node as zero.

2. **Traversal:**

   - While the priority queue is not empty, repeat the following steps:

     - Dequeue a node with the lowest path cost from the priority queue. This node represents the current node being explored.

- Visit and process the current node.

- For each unvisited neighboring node of the current node:

  - Calculate the total cost to reach that neighboring node via the current node.

  - Enqueue the neighboring node into the priority queue with its total cost.

  - Update the cost to reach the neighboring node if a lower-cost path is found.

3. **Termination:**

 - Terminate the algorithm when the priority queue becomes empty or when the goal node is reached.

Key Points:

- UCS explores nodes with the lowest total cost first, ensuring that the shortest path to each explored node is found.

- It's optimal for finding the shortest path in weighted graphs.

- UCS can handle graphs with varying edge costs and guarantees the shortest path to each explored node.

Code:

```
from collections import deque


def main():
    n = int(input("Enter the number of nodes:"))
    start = int(input("Enter the source node: "))
    end = int(input("Enter the ending node: "))
    print("Enter the edges of tree:")
    tree = [[] for _ in range(n+1)]
    for _ in range(n-1):
```

```
    u, v = map(int, input().split())
    tree[u].append(v)
    tree[v].append(u)

parent = [-1] * (n+1)
cost = [0] * (n+1)
closelist = []
q = deque()
found = False

q.append(start)
while q:
    currNode = q.popleft()
    print('\nCurrent Node:',currNode)
    print('ClosedList:',closelist)
    closelist.append(currNode)

    if currNode == end:
        found = True
        break

    for neighbor in tree[currNode]:
        if neighbor in closelist:
            continue

        parent[neighbor] = currNode
        cost[neighbor] = cost[currNode] + 1
        q.append(neighbor)
```

```
    if not found:
        print("Goal node not found!!!")
        return

    print("\nGoal node found!!!")
    path = []
    end_copy = end
    while end_copy != -1:
        path.append(end_copy)
        end_copy = parent[end_copy]
    path.reverse()

    print("\nPath Cost:", cost[end])
    print("\nPath:", end=" ")
    for node in range(len(path)):
        print(path[node], end=(" " if node == len(path)-1 else " -> "))
    print()

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

Output:

Output

```
Enter the number of nodes:7
Enter the source node: 1
Enter the ending node: 6
Enter the edges of tree:
1 2
1 3
2 4
2 5
3 6
3 7

Current Node: 1
ClosedList: []

Current Node: 2
ClosedList: [1]

Current Node: 3
ClosedList: [1, 2]

Current Node: 4
ClosedList: [1, 2, 3]

Current Node: 5
ClosedList: [1, 2, 3, 4]

Current Node: 6
ClosedList: [1, 2, 3, 4, 5]

Goal node found!!!

Path Cost: 2

Path: 1 -> 3 -> 6

=== Code Execution Successful ===
```