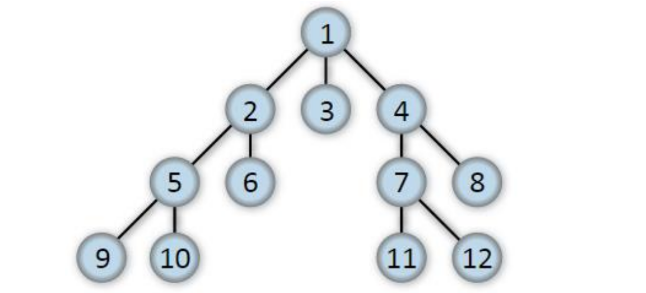
**Task 1: Breadth First Search (BFS)**

**Objective:**

Implement a breadth-first search function that finds and returns the path from the start node to the goal node in a given tree, represented as a graph.

**Task Name:**

Construct the given tree and write a breadth-first search function breath\_first\_search(graph, start\_node, goal\_node) which should return the path to the goal node if the data is found in the tree, else return None.



**Code:**

def breath\_first\_search(graph, start\_node, goal\_node):

visited = set()

queue = [[start\_node]] # Initialize a queue with the start node in a list

if start\_node == goal\_node:

return [start\_node]

while queue:

path = queue.pop(0)

node = path[-1]

if node not in visited:

visited.add(node)

for neighbour in graph[node]:

new\_path = list(path)

new\_path.append(neighbour

queue.append(new\_path)

if neighbour == goal\_node:

return new\_path

return None

graph = {

1: [2, 3, 4],

2: [5, 6],

3: [],

4: [7, 8],

5: [9, 10],

6: [],

7: [11, 12],

8: [],

9: [],

10: [],

11: [],

12: []

}

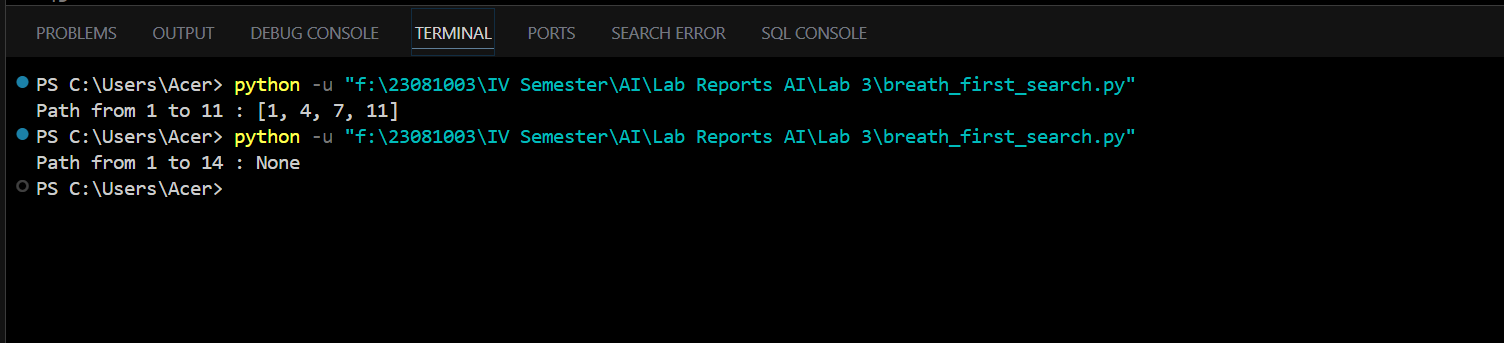
start\_node = 1

goal\_node = 11

path = breath\_first\_search(graph, start\_node, goal\_node)

print("Path from", start\_node, "to", goal\_node, ":", path)

**Output:**

****

**Task 2: Depth First Search (DFS)**

**Objective:**

Implement a depth-first search function that finds and returns the path from the start node to the goal node in a given tree, represented as a graph.

**Task Name:**

Construct the tree as in the previous question, write a depth-first search function depth\_first\_search(graph, start\_node, goal\_node) which should return the path to the goal node if the data is found in the tree, else return None.

**Code:**

def depth\_first\_search(graph, start\_node, goal\_node):

visited = set()

stack = [[start\_node]]

if start\_node == goal\_node:

return [start\_node]

while stack:

path = stack.pop()

node = path[-1]

if node not in visited:

visited.add(node)

for neighbour in graph[node]:

new\_path = list(path)

new\_path.append(neighbour)

stack.append(new\_path)

if neighbour == goal\_node:

return new\_path

return None

graph = {

1: [2, 3, 4],

2: [5, 6],

3: [],

4: [7, 8],

5: [9, 10],

6: [],

7: [11, 12],

8: [],

9: [],

10: [],

11: [],

12: []

}

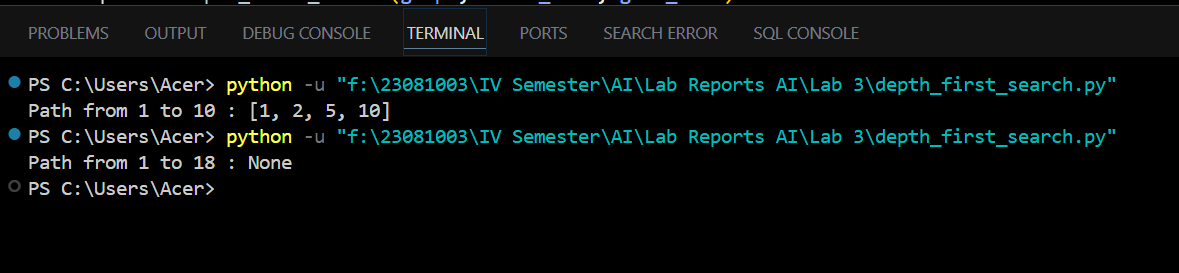
start\_node = 1

goal\_node =

path = depth\_first\_search(graph, start\_node, goal\_node)

print("Path from", start\_node, "to", goal\_node, ":", path)

**Output:**

****

**Conclusion:**

In this assignment, we implemented both breadth-first search (BFS) and depth-first search (DFS) algorithms in Python to traverse a tree structure and find a path from the start node to the goal node. The BFS function efficiently explored the shortest path in a level-wise manner, ensuring that all nodes at the present depth were checked before moving on to nodes at the next depth level. On the other hand, the DFS function delved deep into the tree, exploring as far as possible along each branch before backtracking, which is useful for scenarios requiring deep path explorations. We defined a graph structure and utilized sets to keep track of visited nodes to prevent reprocessing the same nodes, thus optimizing the search process. By using Python, we leveraged its powerful data structures, such as lists and dictionaries, to efficiently manage the queue and stack operations needed for BFS and DFS respectively. This exercise was our first experience using Python for implementing graph traversal algorithms, and it demonstrated the language's capability in handling complex data structures and algorithms with simplicity and readability.