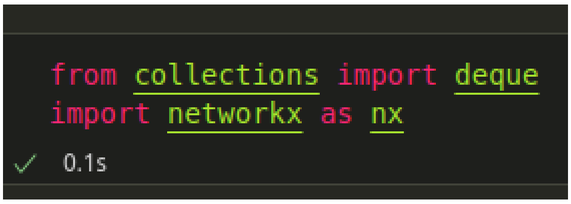
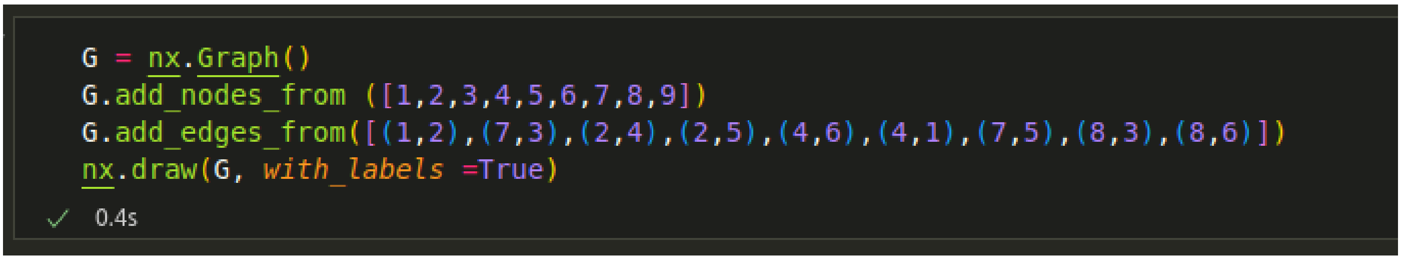
ASSIGNMENT – 1

The code implements a function to find the shortest path between two nodes in an unweighted graph using the *Breadth-First Search (BFS)* algorithm. Here's a summary of the key components and their roles:



```
from collections import deque
import networkx as nx
✓ 0.1s
```

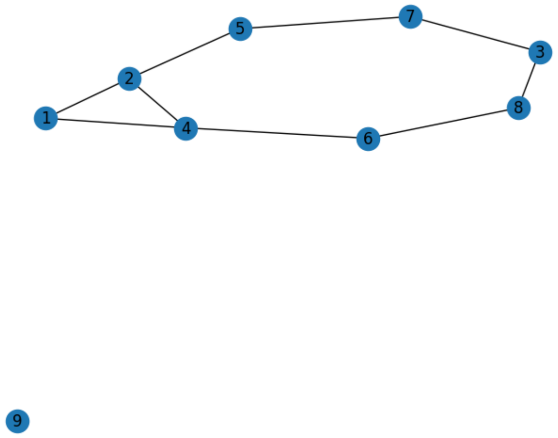
- I have used a library called ‘ *NetworkX’* for creating the network and visualizing the nodes.

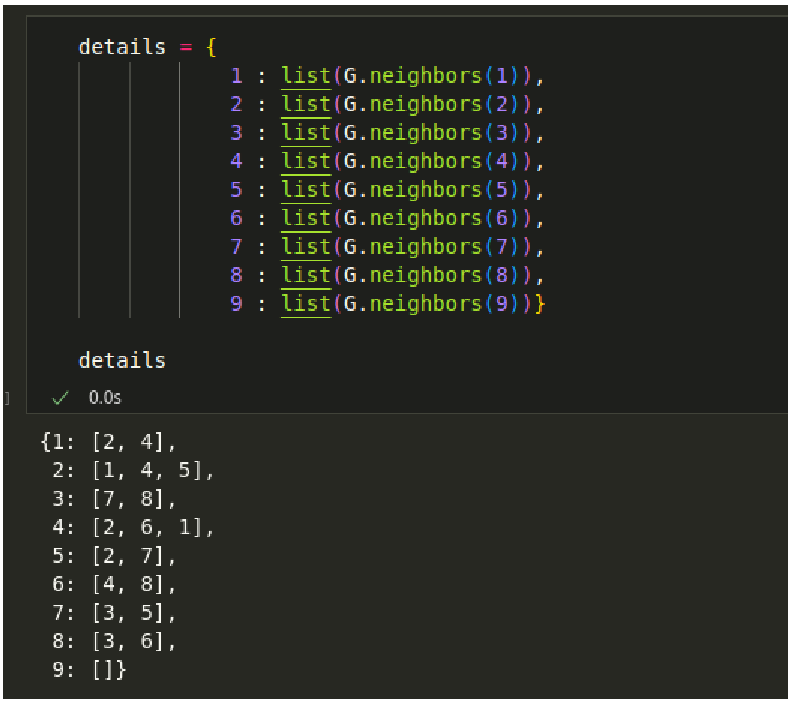


```
G = nx.Graph()
G.add_nodes_from ([1,2,3,4,5,6,7,8,9])
G.add_edges_from([(1,2),(7,3),(2,4),(2,5),(4,6),(4,1),(7,5),(8,3),(8,6)])
nx.draw(G, with_labels =True)
✓ 0.4s
```

- A graph object is created, and nodes are added.
- Defined the edges between the nodes.

Output :



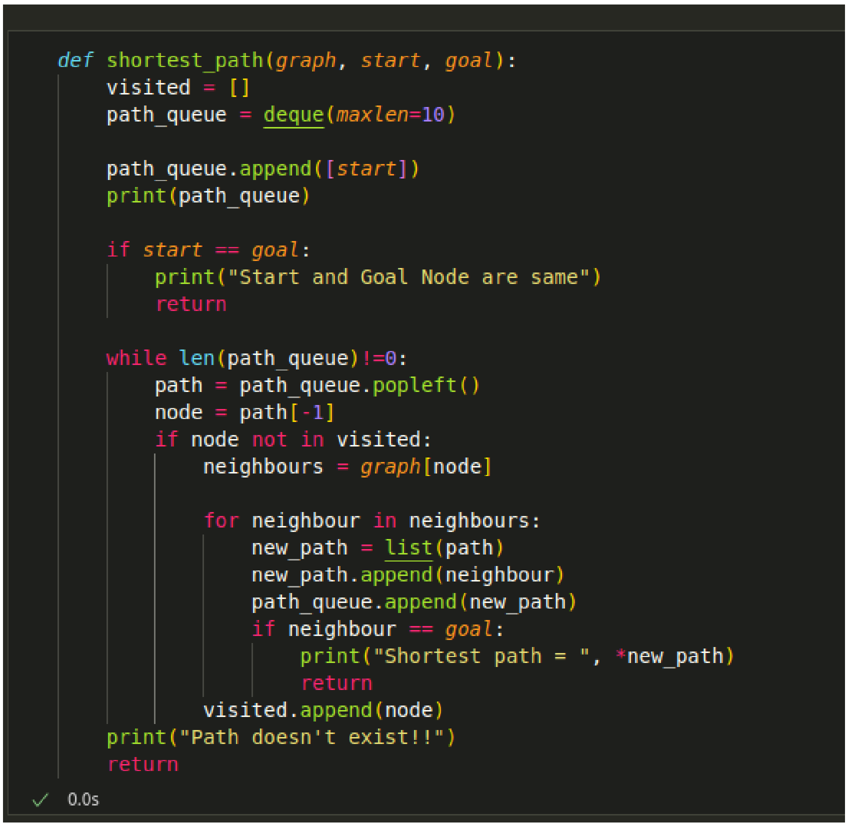


```
details = {
            1 : list(G.neighbors(1)),
            2 : list(G.neighbors(2)),
            3 : list(G.neighbors(3)),
            4 : list(G.neighbors(4)),
            5 : list(G.neighbors(5)),
            6 : list(G.neighbors(6)),
            7 : list(G.neighbors(7)),
            8 : list(G.neighbors(8)),
            9 : list(G.neighbors(9))}

details
```

```
{1: [2, 4],
 2: [1, 4, 5],
 3: [7, 8],
 4: [2, 6, 1],
 5: [2, 7],
 6: [4, 8],
 7: [3, 5],
 8: [3, 6],
 9: []}
```

- Created a Dictionary Object where every node (key) is matched with its neighbors.

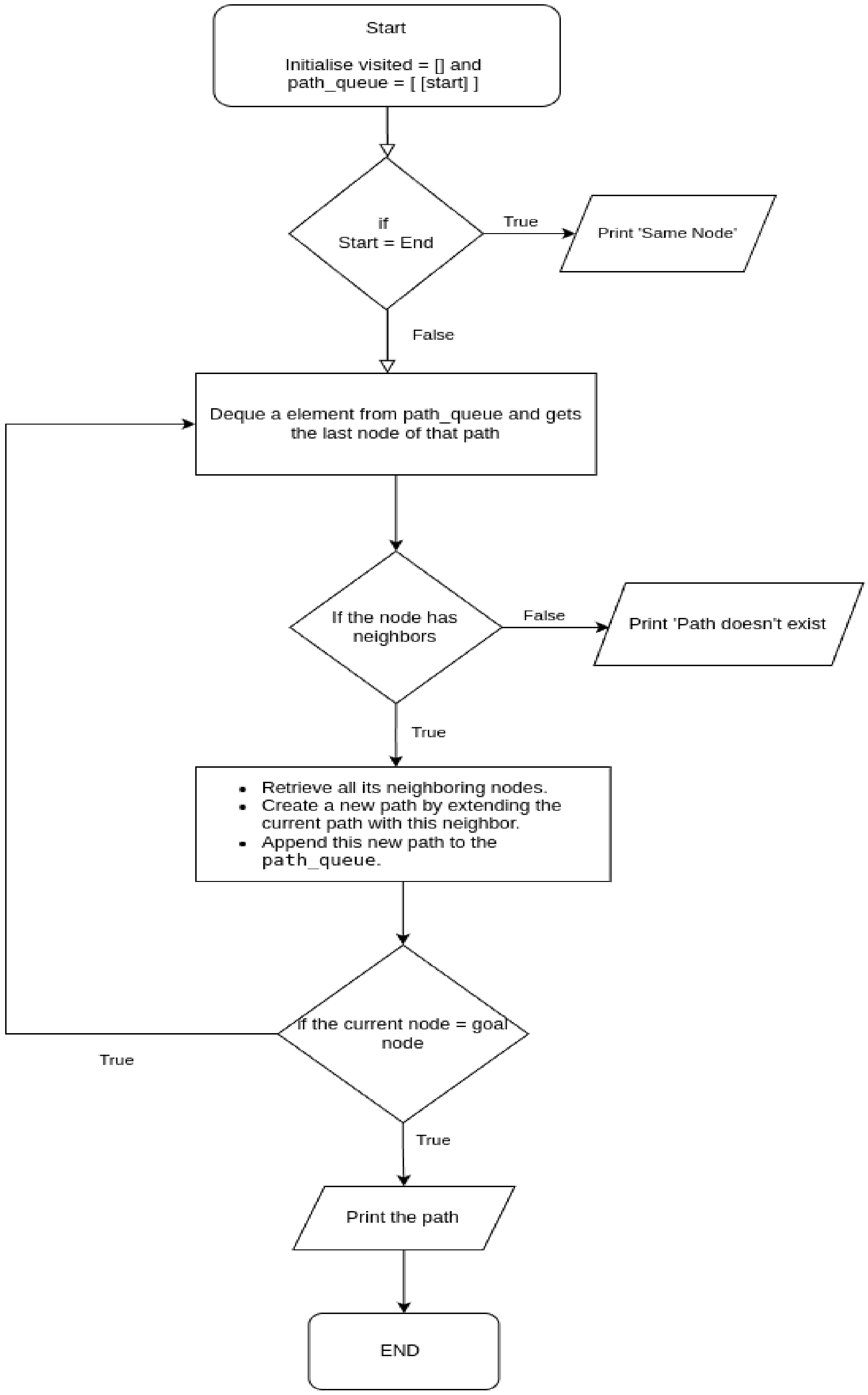
ALGORITHM:



```
def shortest_path(graph, start, goal):
    visited = []
    path_queue = deque(maxlen=10)

    path_queue.append([start])
    print(path_queue)

    if start == goal:
        print("Start and Goal Node are same")
        return

    while len(path_queue)!=0:
        path = path_queue.popleft()
        node = path[-1]
        if node not in visited:
            neighbours = graph[node]

            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                path_queue.append(new_path)
                if neighbour == goal:
                    print("Shortest path = ", *new_path)
                    return
            visited.append(node)
    print("Path doesn't exist!!")
    return
```

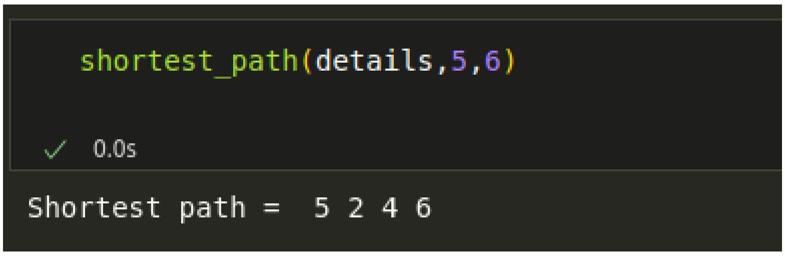
What the Algorithm Does ?

- The function takes neighbor details(dictionary), start and goal node as input arguments.


Start
Initialise visited = [] and
path_queue = [[start]]
if
Start = End
True
Print 'Same Node'
False
Deque a element from path_queue and gets the last node of that path
If the node has neighbors
False
Print 'Path doesn't exist
True
Retrieve all its neighboring nodes.
Create a new path by extending the current path with this neighbor.
Append this new path to the path_queue.
if the current node = goal node
True
True
Print the path
END

- The use of a queue ensures that paths are explored in the correct order (FIFO - First In, First Out), which is crucial for the BFS algorithm.
- The algorithm traverses the graph level by level. It explores all nodes at the present depth level before moving on to nodes at the next depth level.
-
- This level-order traversal ensures that the first time it reaches the goal node, it does so via the shortest path.

<u>OUTPUT:</u>



```
shortest_path(details,5,6)
```

✓ 0.0s

```
Shortest path =  5 2 4 6
```

- The output can be verified by checking the graph that is displayed above.