EXPERIMENT NO. 2

Student Name and Roll Number: Piyush Gambhir – 21CSU349	
Semester /Section: Semester-V – AIML-V-B (AL-3)	
Link to Code: NCU-Lab-Manual-And-End-Semester-Projects/NCU-CSL347-AAIES-Lab_Manual at main · Piyush-Gambhir/NCU-Lab-Manual-And-End-Semester-Projects (github.com)	
Date: 19.08.2023	
Faculty Signature:	
Grade:	

Objective(s):

- Understand what breadth first Search (BFS) and Depth first Search (DFS) is.
- Study about different uninformed searching approaches.
- Implement BFS and DFS for solving a real-world problem.

Outcome:

Students would be familiarized with BFS and DFS.
Students would be able to make a comparison between the two algorithms.

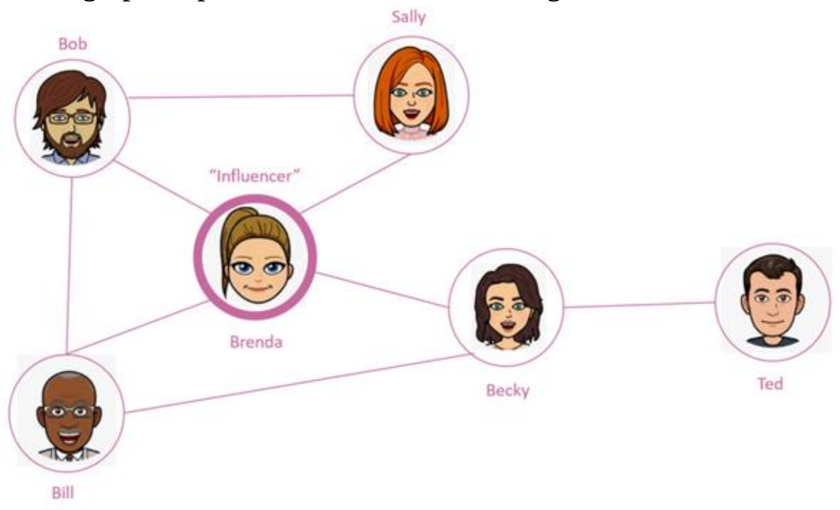
Problem Statement:

Implement Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in Python to analyze a simple social network. The program aims to explore social connections between users and offer insights into their relationships.

The program should be able to:

- Find groups of users who are directly or indirectly connected to the influencer "*Brenda*".
- Determine if there is a path between two users "*Sally*" and "*Ted*"

Use the graph as provided below for the assignment:



Background Study:

Breadth-First Search (BFS) and Depth-First Search (DFS) are graph traversal algorithms. BFS explores a graph level by level, visiting all neighboring nodes before moving deeper, utilizing a queue data structure. On the other hand, DFS explores as deep as possible along each branch before backtracking, using a stack data structure. Both algorithms are fundamental in graph analysis and have various applications in tasks like path finding, cycle detection, and social network analysis.

Question Bank:

1. What do you understand by blind search algorithms?
   Blind search algorithms are methods used in computer science to explore and traverse problem spaces without having any specific information about the structure or characteristics of the space. These algorithms rely solely on the information available during the search process and do not incorporate domain-specific knowledge.

2. What is *Breadth-First Search* and *Depth-First Search*?
   Blind search algorithms are methods used in computer science to explore and traverse problem spaces without having any specific information about the structure or characteristics of the space. These algorithms rely solely on the information available during the search process and do not incorporate domain-specific knowledge.

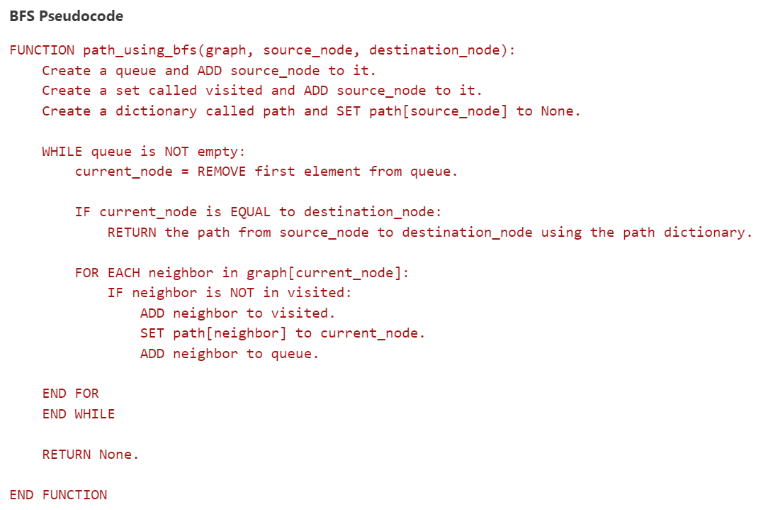
3. What are the appropriate scenarios for using BFS and DFS algorithms?
   Breadth-First Search (BFS) is suitable for scenarios where finding the shortest path is important, like in navigation systems or puzzle-solving. It guarantees the shortest path but can require more memory due to the need to store the entire level's nodes.

   Depth-First Search (DFS) is appropriate for scenarios where memory efficiency is crucial, such as solving mazes or searching through large trees. It doesn't guarantee the shortest path and can get stuck in infinite loops in some cases, but it's memory efficient as it explores one branch deeply before backtracking.

Student Work Area

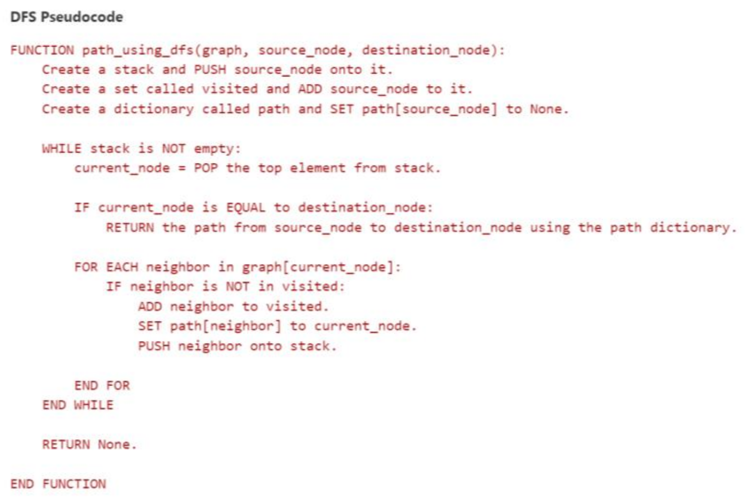
Algorithm/Flowchart/Code/Sample Outputs

Pseudocode



BFS Pseudocode

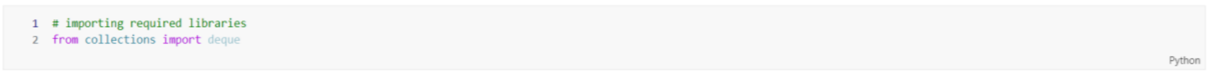
```
FUNCTION path_using_bfs(graph, source_node, destination_node):
    Create a queue and ADD source_node to it.
    Create a set called visited and ADD source_node to it.
    Create a dictionary called path and SET path[source_node] to None.

    WHILE queue is NOT empty:
        current_node = REMOVE first element from queue.

        IF current_node is EQUAL to destination_node:
            RETURN the path from source_node to destination_node using the path dictionary.

        FOR EACH neighbor in graph[current_node]:
            IF neighbor is NOT in visited:
                ADD neighbor to visited.
                SET path[neighbor] to current_node.
                ADD neighbor to queue.

    END FOR
    END WHILE

    RETURN None.

END FUNCTION
```



DFS Pseudocode

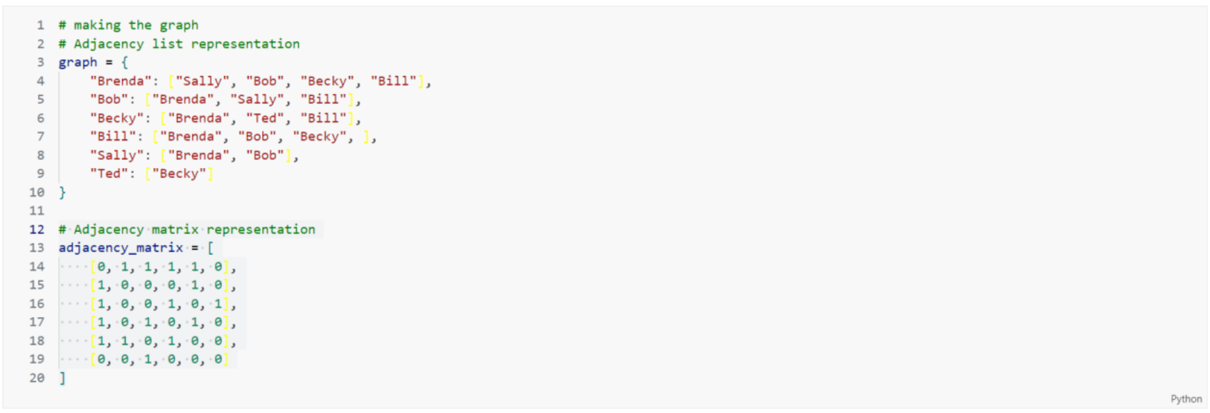
```
FUNCTION path_using_dfs(graph, source_node, destination_node):
    Create a stack and PUSH source_node onto it.
    Create a set called visited and ADD source_node to it.
    Create a dictionary called path and SET path[source_node] to None.

    WHILE stack is NOT empty:
        current_node = POP the top element from stack.

        IF current_node is EQUAL to destination_node:
            RETURN the path from source_node to destination_node using the path dictionary.

        FOR EACH neighbor in graph[current_node]:
            IF neighbor is NOT in visited:
                ADD neighbor to visited.
                SET path[neighbor] to current_node.
                PUSH neighbor onto stack.

        END FOR
    END WHILE

    RETURN None.

END FUNCTION
```

Code:

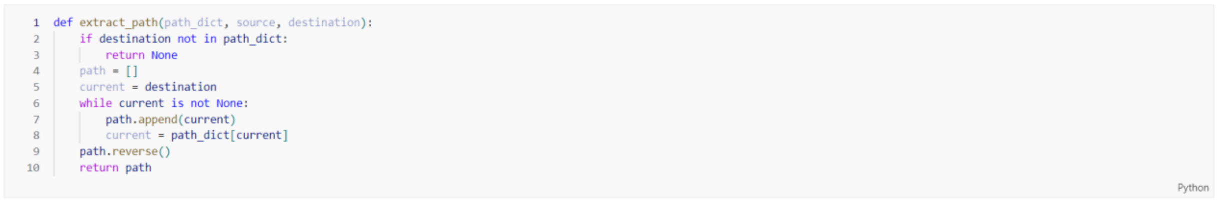


```python
# importing required libraries
from collections import deque
```

Defining the Graph Using Adjacency List

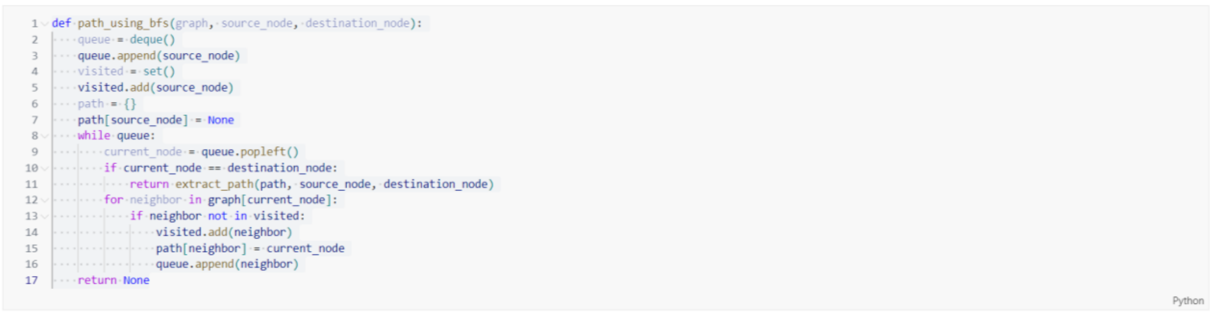


```python
# making the graph
# Adjacency list representation
graph = {
    "Brenda": ["Sally", "Bob", "Becky", "Bill"],
    "Bob": ["Brenda", "Sally", "Bill"],
    "Becky": ["Brenda", "Ted", "Bill"],
    "Bill": ["Brenda", "Bob", "Becky", ],
    "Sally": ["Brenda", "Bob"],
    "Ted": ["Becky"]
}

# Adjacency matrix representation
adjacency_matrix = [
    [0, 1, 1, 1, 1, 0],
    [1, 0, 0, 0, 1, 0],
    [1, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0],
    [1, 1, 0, 1, 0, 0],
    [0, 0, 1, 0, 0, 0]
]
```



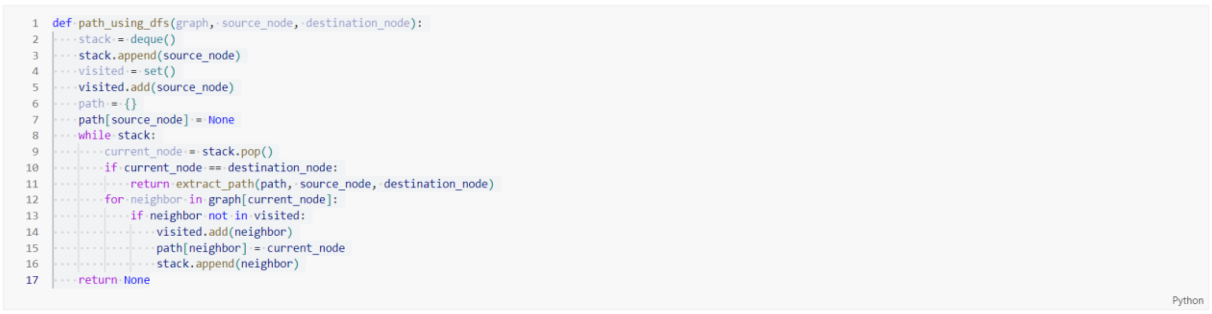
```python
def extract_path(path_dict, source, destination):
    if destination not in path_dict:
        return None
    path = []
    current = destination
    while current is not None:
        path.append(current)
        current = path_dict[current]
    path.reverse()
    return path
```

Path Using BFS



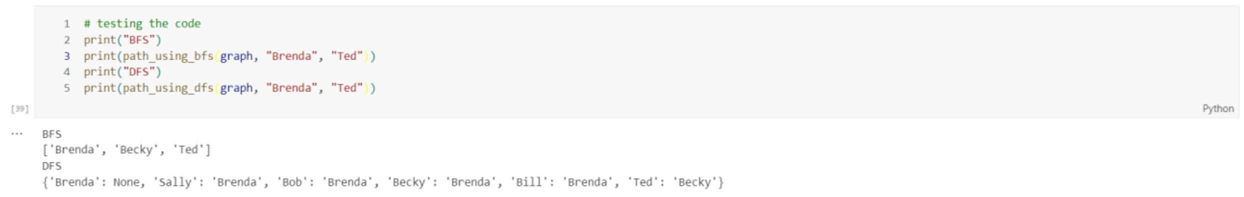
```python
def path_using_bfs(graph, source_node, destination_node):
    queue = deque()
    queue.append(source_node)
    visited = set()
    visited.add(source_node)
    path = {}
    path[source_node] = None
    while queue:
        current_node = queue.popleft()
        if current_node == destination_node:
            return extract_path(path, source_node, destination_node)
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                path[neighbor] = current_node
                queue.append(neighbor)
    return None
```

Path Using DFS



```python
def path_using_dfs(graph, source_node, destination_node):
    stack = deque()
    stack.append(source_node)
    visited = set()
    visited.add(source_node)
    path = {}
    path[source_node] = None
    while stack:
        current_node = stack.pop()
        if current_node == destination_node:
            return extract_path(path, source_node, destination_node)
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                path[neighbor] = current_node
                stack.append(neighbor)
    return None
```

Output:



```python
# testing the code
print("BFS")
print(path_using_bfs(graph, "Brenda", "Ted"))
print("DFS")
print(path_using_dfs(graph, "Brenda", "Ted"))
```

```
BFS
['Brenda', 'Becky', 'Ted']
DFS
{'Brenda': None, 'Sally': 'Brenda', 'Bob': 'Brenda', 'Becky': 'Brenda', 'Bill': 'Brenda', 'Ted': 'Becky'}
```