THE NORTHCAP UNIVERSITY

EXPERIMENT NO. 2

Student Name and Roll Number: Shivam / 21CSU090
Semester /Section: V / AIML – A1
Link to Code:
Date: 19/8/2023
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.

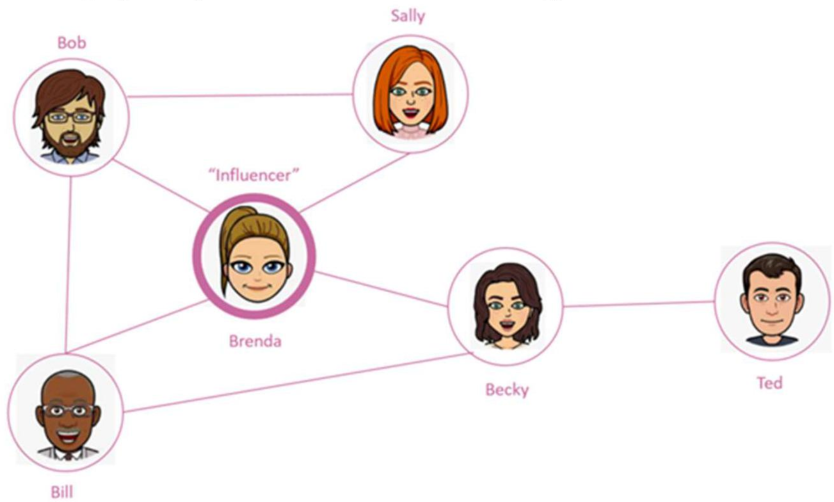
Problem Statement:

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?
Ans: Blind search algorithms, also known as uninformed search algorithms, are search strategies that explore a problem space without exploiting specific information about the problem's nature. These algorithms rely solely on the structure of the search space and do not use any heuristics or domain-specific knowledge to guide their search. Blind search algorithms traverse the search space systematically, often using a set of rules to determine the order of exploration.

2. What is *Breadth-First Search* and *Depth-First Search*?
Ans:

- BFS explores level by level, using a queue to maintain order.
- DFS explores deeply, using recursion or a stack for backtracking.
- BFS is suitable for finding shortest paths and analyzing connected components.
- DFS is used for tasks like maze solving, topological sorting, and deep exploration.

3. What are the appropriate scenarios for using BFS and DFS algorithms?

Ans:

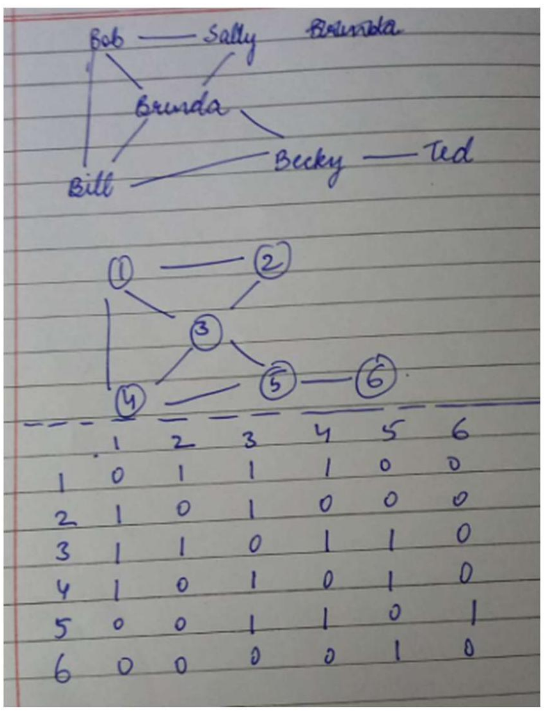
- BFS is chosen when finding shortest paths is essential, systematic exploration is needed, and solution space is shallow.
- DFS is preferred for deep exploration, non-shortest path goals, memory-efficient searches, and pattern detection tasks.

Student Work Area

Algorithm/Flowchart/Code/Sample Outputs

```
function bfs_find_path(graph, source_node, destination_node):
    if source_node not in graph or destination_node not in graph:
        return None

    create an empty queue
    enqueue (source_node, []) to the queue
    create an empty set called visited

    while queue is not empty:
        dequeue a node and its path from the queue

        if current_node is equal to destination_node:
            return path + [current_node]

        add current_node to visited

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                enqueue (neighbor, path + [current_node]) to the queue

    return None
--------------------------------------------------------------------------------

function dfs_find_path(graph, current_node, destination_node, visited, path):
    if current_node is destination_node:
        return path + [current_node]
    add current_node to visited
```

```
    add current_node to path
    for neighbor in graph[current_node]:
        if neighbor not in visited:
            new_path = dfs_find_path(graph, neighbor, destination_node, visited +
[current_node], path + [current_node])
            if new_path is not None:
                return new_path

    return None
```



	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	1	0	0	0
3	1	1	0	1	1	0
4	1	0	1	0	1	0
5	0	0	1	1	0	1
6	0	0	0	0	1	0

Imports Needed

```python
from collections import defaultdict, deque
```

[1] 0.0s

Define the graph using adjacency list

```python
# A graph of friends
graph = {
    "Bob": ["Bill", "Brenda", "Sally"],
    "Bill": ["Bob", "Brenda", "Becky"],
    "Brenda": ["Bill", "Bob", "Becky", "Sally"],
    "Becky": ["Brenda", "Ted", "Bill"],
    "Ted": ["Becky"],
    "Sally": ["Brenda", "Bob"]
}
```

[10] 0.0s

```python
# BFS: breadth-first search
# BFS can be used to find the shortest path between two nodes in an unweighted graph.

def path_using_bfs(graph, source_node, destination_node):
    """Finds a path from source_node to destination_node in the given graph."""
    if source_node not in graph or destination_node not in graph:
        return None  # One of the nodes is not in the graph

    visited = set()
    queue = deque([(source_node, [])])  # Queue stores nodes and their paths

    while queue:
        current_node, path = queue.popleft()
        visited.add(current_node)

        for neighbor in graph[current_node]:
            if neighbor == destination_node:
                return path + [current_node, neighbor]  # Path found
            if neighbor not in visited:
                queue.append((neighbor, path + [current_node]))

    return None  # No path found

source_node = input("Enter the source node: ")
destination_node = input("Enter the destination node: ")
path = path_using_bfs(graph, source_node, destination_node)

if path:
    print(f"Path from {source_node} to {destination_node}: {path}")
else:
    print(f"No path found from {source_node} to {destination_node}")
```

[11] ✓ 3.8s

```
Path from Sally to Ted: ['Sally', 'Brenda', 'Becky', 'Ted']
```

```python
def path_using_dfs(graph, current_node, destination_node, visited, path):
    """Returns a list of nodes in a path from current_node to destination_node in graph."""
    visited.add(current_node)
    path.append(current_node)

    if current_node == destination_node:
        return path

    for neighbor in graph[current_node]:
        if neighbor not in visited:
            new_path = path_using_dfs(graph, neighbor, destination_node, visited.copy(), path.copy())
            if new_path:
                return new_path

    return None

source_node = input("Enter the source node: ")
destination_node = input("Enter the destination node: ")
path = path_using_dfs(graph, source_node, destination_node, set(), [])

if path:
    print(f"Path from {source_node} to {destination_node}: {path}")
else:
    print(f"No path found from {source_node} to {destination_node}")
```

3.7s

```
Path from Sally to Ted: ['Sally', 'Brenda', 'Bill', 'Becky', 'Ted']
```