UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Vasig, Yuan Hessed O.

Instructor:
Engr. Maria Rizette H. Sayo

October, 18, 2025

I. Objectives

Introduction

A graph is a visual representation of a collection of things where some object pairs are linked together. Vertices are the points used to depict the interconnected items, while edges are the connections between them. In this course, we go into great detail on the many words and functions related to graphs.

An undirected graph, or simply a graph, is a set of points with lines connecting some of the points. The points are called nodes or vertices, and the lines are called edges.

A graph can be easily presented using the python dictionary data types. We represent the vertices as the keys of the dictionary and the connection between the vertices also called edges as the values in the dictionary.

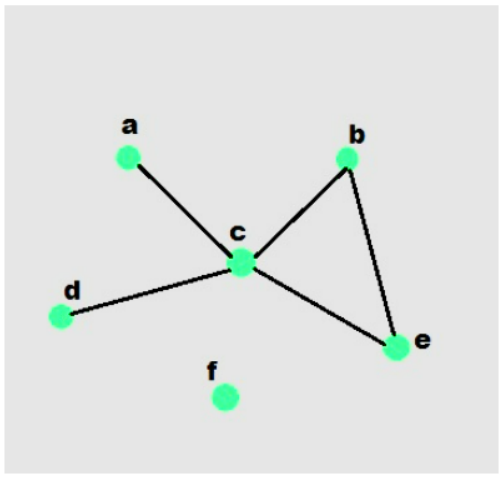


Figure 1. Sample graph with vertices and edges

This laboratory activity aims to implement the principles and techniques in:

- To introduce the Non-linear data structure – Graphs
- To implement graphs using Python programming language
- To apply the concepts of Breadth First Search and Depth First Search

II. Methods

A. Copy and run the Python source codes.
B. If there is an algorithm error/s, debug the source codes.
C. Save these source codes to your GitHub.

```
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        """Add an edge between u and v"""
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []

        self.graph[u].append(v)
        self.graph[v].append(u)  # For undirected graph

    def bfs(self, start):
        """Breadth-First Search traversal"""
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)
                # Add all unvisited neighbors
                for neighbor in self.graph.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return result

    def dfs(self, start):
        """Depth-First Search traversal"""
        visited = set()
        result = []

        def dfs_util(vertex):
            visited.add(vertex)
            result.append(vertex)
            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    dfs_util(neighbor)

        dfs_util(start)
        return result

    def display(self):
        """Display the graph"""
        for vertex in self.graph:
            print(f"{vertex}: {self.graph[vertex]}")

# Example usage
if __name__ == "__main__":
    # Create a graph
```

```
g = Graph()

# Add edges
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)

# Display the graph
print("Graph structure:")
g.display()

# Traversal examples
print(f"\nBFS starting from 0: {g.bfs(0)}")
print(f"DFS starting from 0: {g.dfs(0)}")

# Add more edges and show
g.add_edge(4, 5)
g.add_edge(1, 4)

print(f"\nAfter adding more edges:")
print(f"BFS starting from 0: {g.bfs(0)}")
print(f"DFS starting from 0: {g.dfs(0)}")
```

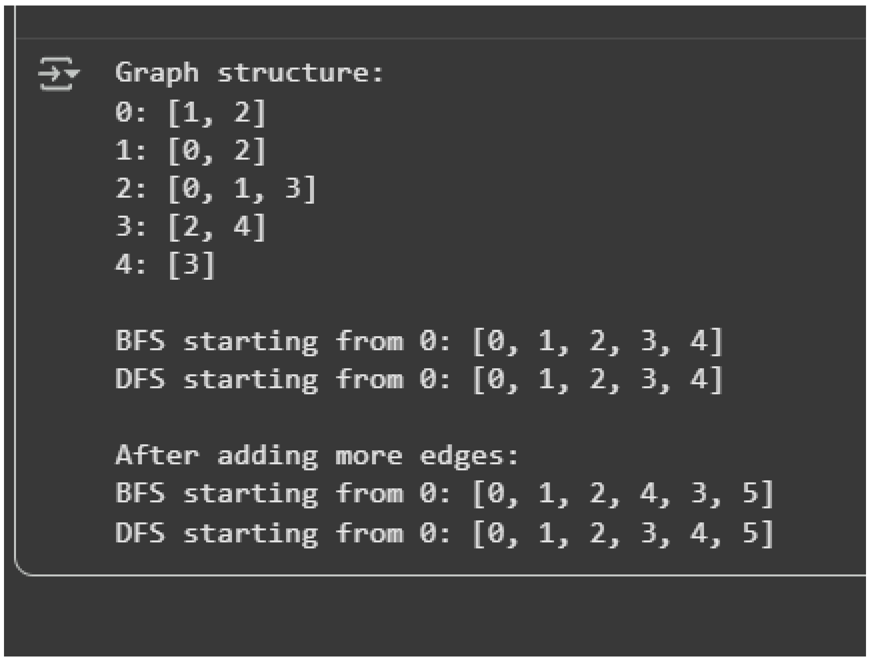
Questions:

1. What will be the output of the following codes?
2. Explain the key differences between the BFS and DFS implementations in the provided graph code. Discuss their data structures, traversal patterns, and time complexity. How does the recursive nature of DFS contrast with the iterative approach of BFS, and what are the potential advantages and disadvantages of each implementation strategy?
3. The provided graph implementation uses an adjacency list representation with a dictionary. Compare this approach with alternative representations like adjacency matrices or edge lists.
4. The graph in the code is implemented as undirected. Analyze the implications of this design choice on the add_edge method and the overall graph structure. How would you modify the code to support directed graphs? Discuss the changes needed in edge addition, traversal algorithms, and how these modifications would affect the graph's behavior and use cases.
5. Choose two real-world problems that can be modeled using graphs and explain how you would use the provided graph implementation to solve them. What extensions or modifications would be necessary to make the code suitable for these applications? Discuss how the BFS and DFS algorithms would be particularly useful in solving these problems and what additional algorithms you might need to implement.

III. Results

Solution

1.



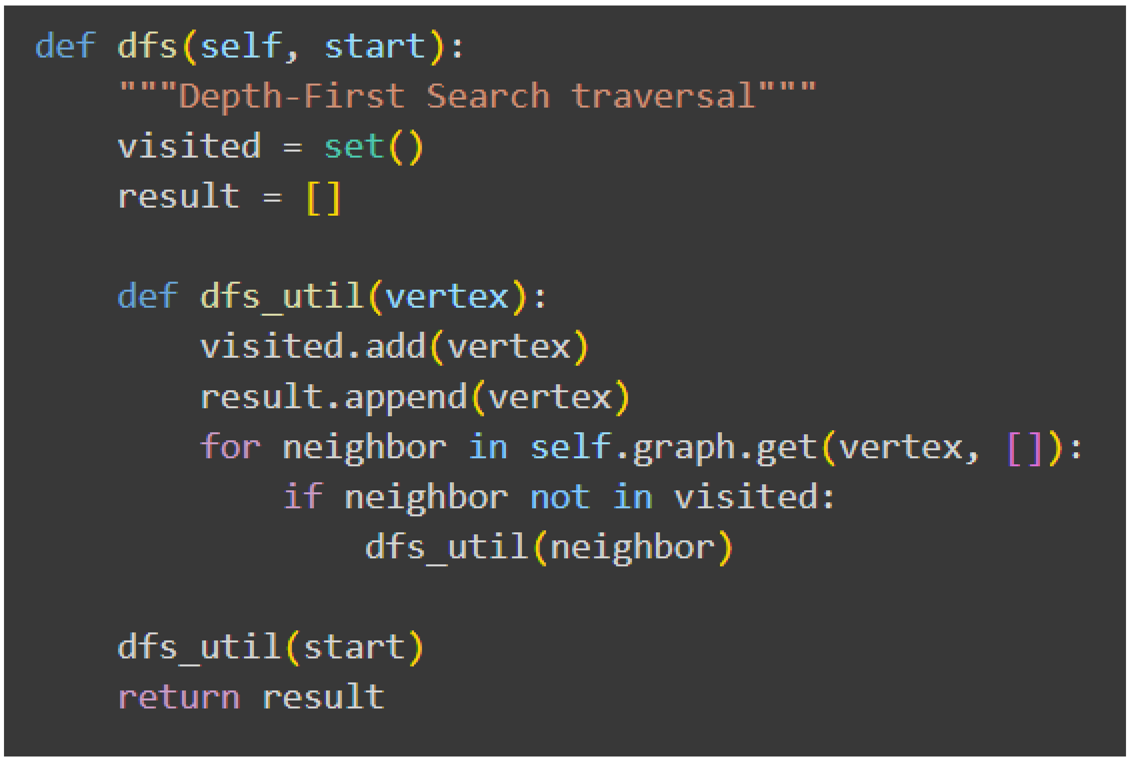
```
Graph structure:
0: [1, 2]
1: [0, 2]
2: [0, 1, 3]
3: [2, 4]
4: [3]

BFS starting from 0: [0, 1, 2, 3, 4]
DFS starting from 0: [0, 1, 2, 3, 4]

After adding more edges:
BFS starting from 0: [0, 1, 2, 4, 3, 5]
DFS starting from 0: [0, 1, 2, 3, 4, 5]
```

Figure 1: Output of the Code

2. Difference Between BFS and DFS Implementation

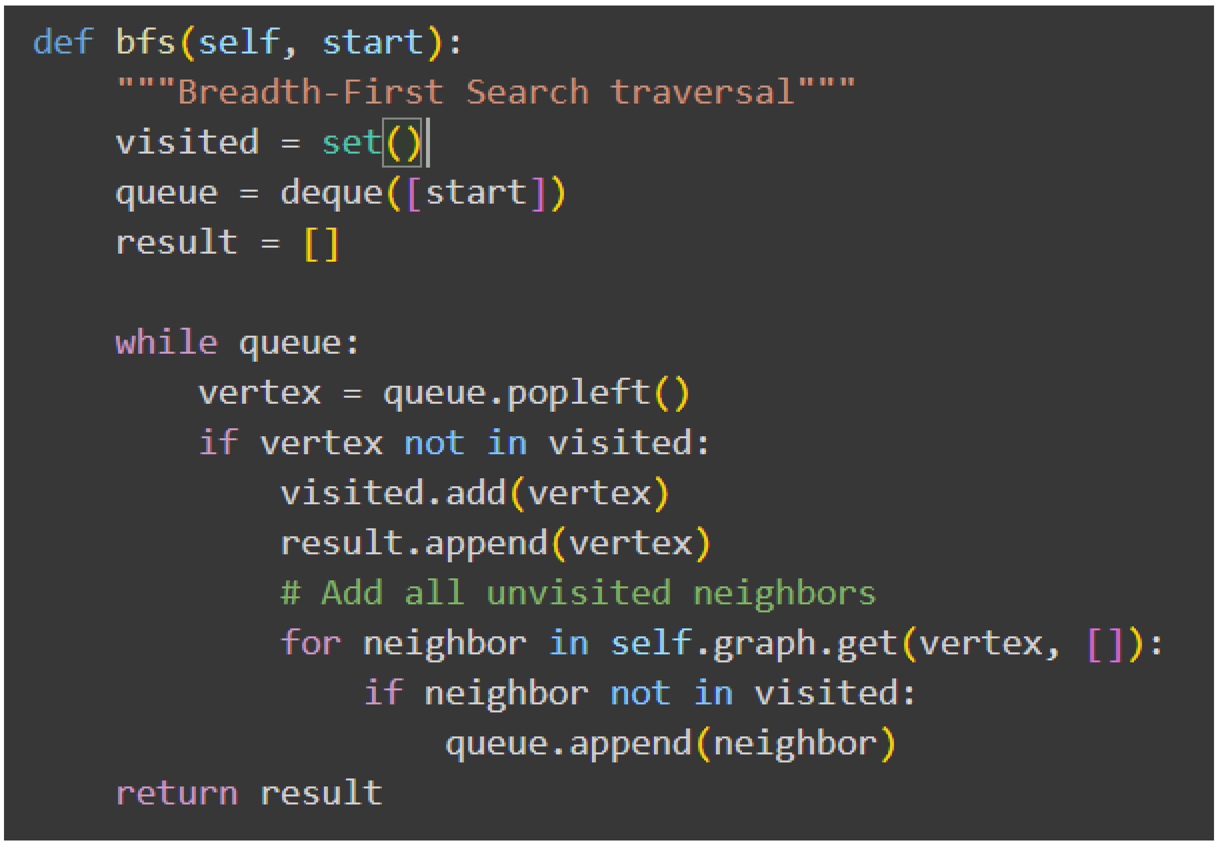


```python
def dfs(self, start):
    """Depth-First Search traversal"""
    visited = set()
    result = []

    def dfs_util(vertex):
        visited.add(vertex)
        result.append(vertex)
        for neighbor in self.graph.get(vertex, []):
            if neighbor not in visited:
                dfs_util(neighbor)

    dfs_util(start)
    return result
```

Figure 2: DFS function used



```
def bfs(self, start):
    """Breadth-First Search traversal"""
    visited = set()
    queue = deque([start])
    result = []

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            result.append(vertex)
            # Add all unvisited neighbors
            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    queue.append(neighbor)
    return result
```

Figure 3: BFS function used

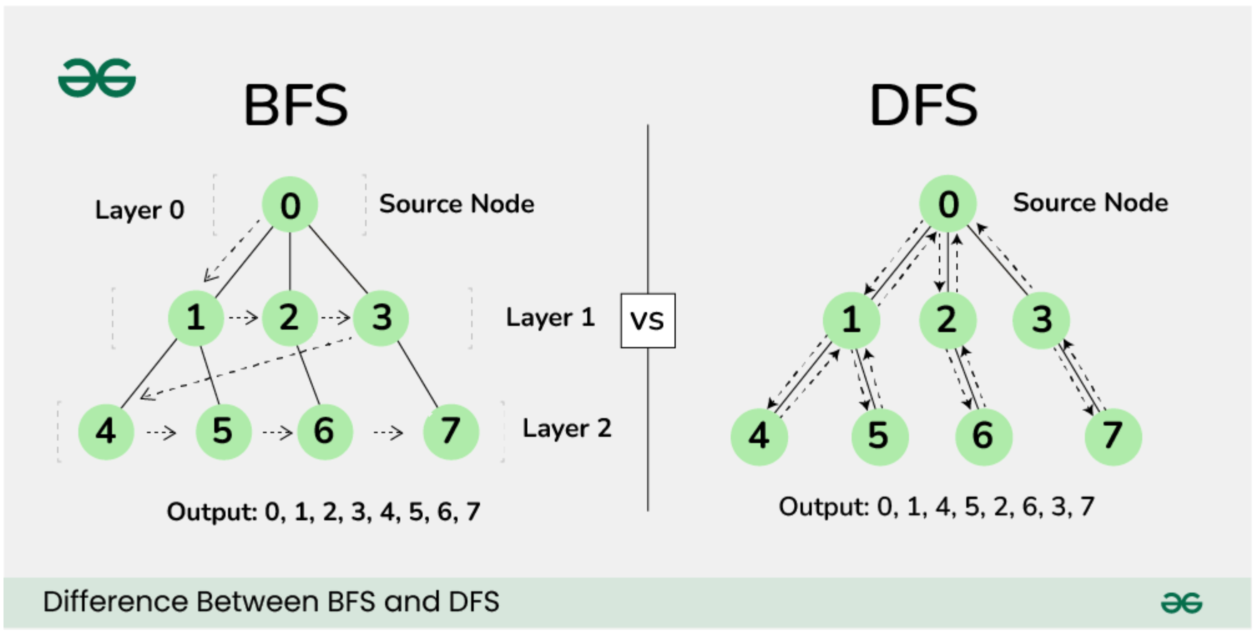
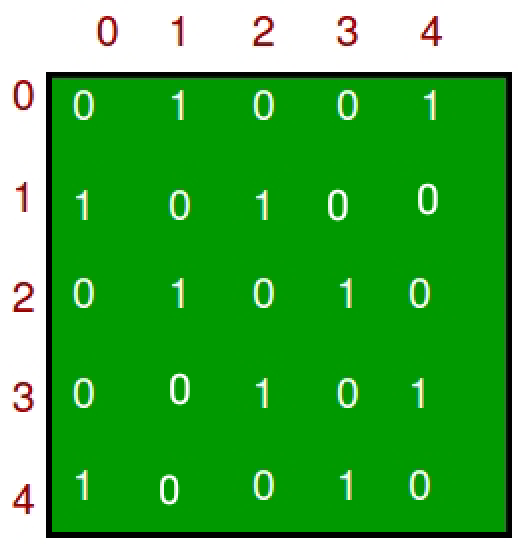

BFS
Layer 0
0
Source Node
1
2
3
Layer 1
4
5
6
7
Layer 2
Output: 0, 1, 2, 3, 4, 5, 6, 7
VS
DFS
0
Source Node
1
2
3
4
5
6
7
Output: 0, 1, 4, 5, 2, 6, 3, 7
Difference Between BFS and DFS

Figure 4: Difference between BFS and DFS

https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

Breadth-First Search (BFS) uses a queue and works iteratively to explore nodes level by level, making it ideal for finding the shortest path in unweighted graphs. Depth-First Search (DFS), on the other hand, uses recursion (which relies on a call stack) to explore one path as deeply as possible before backtracking. Both algorithms have a time complexity of O(V + E), but BFS may require more memory in wide graphs due to its queue, while DFS can risk stack overflow in very deep or recursive graphs. BFS is more systematic and guarantees the shortest path, whereas DFS is better for exploring all possible paths or detecting cycles.

3. Graph Representation Comparison(Adjacency List vs Adjacency Matrix)



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	0
3	0	0	1	0	1
4	1	0	0	1	0

Figure 5: Adjacency Matrix

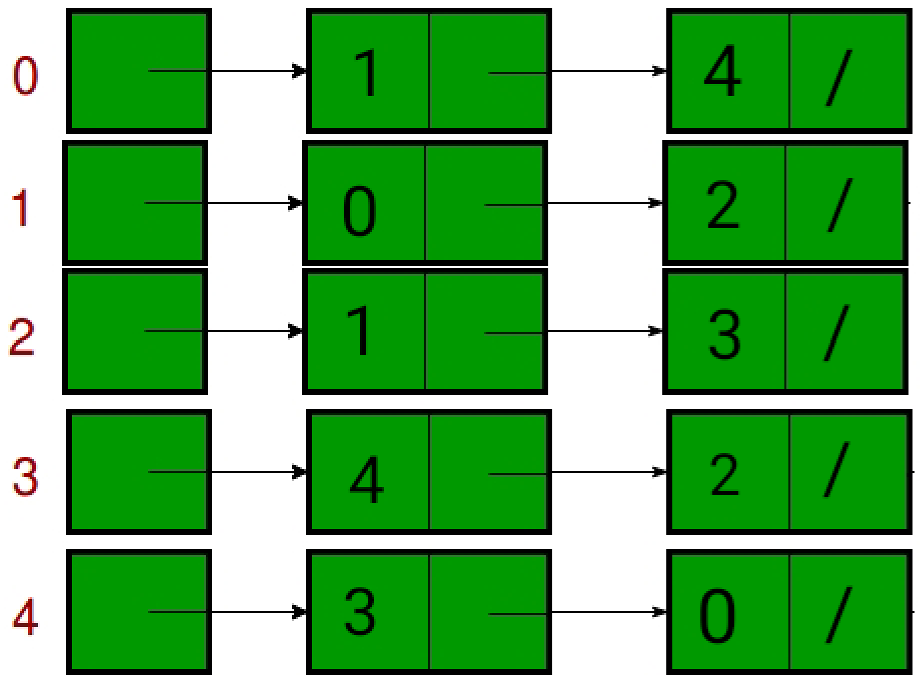
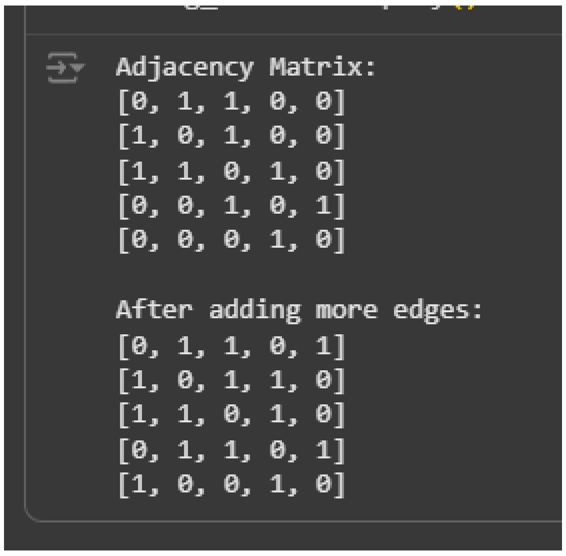


Figure 6:Adjacency List



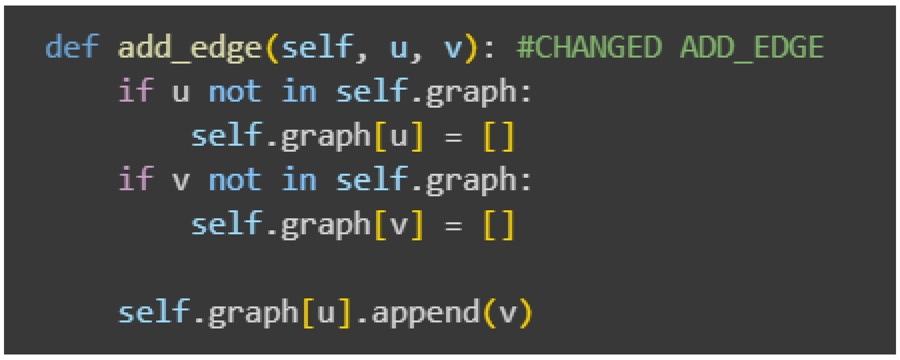
```
Adjacency Matrix:
[0, 1, 1, 0, 0]
[1, 0, 1, 0, 0]
[1, 1, 0, 1, 0]
[0, 0, 1, 0, 1]
[0, 0, 0, 1, 0]

After adding more edges:
[0, 1, 1, 0, 1]
[1, 0, 1, 1, 0]
[1, 1, 0, 1, 0]
[0, 1, 1, 0, 1]
[1, 0, 0, 1, 0]
```

Figure 7: Output for Adjacency Matrix

An adjacency list uses a dictionary or list where each vertex maps to a list of its neighboring vertices, making it very **space-efficient for sparse graphs** and easy to update when adding or removing edges. However, checking whether a specific edge exists can be slower since it requires searching through a vertex's neighbor list. In contrast, an adjacency matrix uses a two-dimensional array where each cell (i, j) indicates whether there is an edge between vertices i and j. This allows **constant-time edge lookups** but consumes **more memory**, as it must store all possible vertex pairs, even when most are not connected. The adjacency matrix is ideal for **dense graphs** or when frequent edge existence checks are needed, while the adjacency list is generally preferred for **sparse or dynamic graphs** and for traversal algorithms like BFS and DFS due to its efficiency in space and neighbor iteration.

4.



```python
def add_edge(self, u, v): #CHANGED ADD_EDGE
    if u not in self.graph:
        self.graph[u] = []
    if v not in self.graph:
        self.graph[v] = []

    self.graph[u].append(v)
```

Figure 8:Structural Change

By modifying the add_edge method to add only a single directed connection (u → v), the graph correctly models these one-directional relationships. This change alters how traversals like BFS and DFS behave: they now follow edges only in their defined direction, which affects which nodes can be reached from a given starting point. In essence, the modification ensures the graph's structure and algorithms accurately reflect the intended directional behavior of the data it represents.

5.

Real-World Problem 1: Social Network Analysis

How it can be modeled

A **social network** can be represented as a **graph** where:

- - Each **user** is a vertex (node).
- Each **friendship** or **follow** relationship is an edge.
- For **friendships** (mutual connections like Facebook), we'd use an **undirected graph**.
- For **followers** (one-way connections like Twitter), we'd use a **directed graph**.

Real-World Problem 2: Network Routing (Internet or Road Networks)

How it can be modeled

A **computer network** or **road system** can also be modeled as a graph:

- **Nodes (vertices):** Routers, computers, or intersections.
- **Edges:** Network links or roads.
- **Weights:** Represent bandwidth, latency, or distance.

IV. Conclusion

In conclusion, this laboratory activity successfully demonstrated the implementation and practical applications of graphs as a non-linear data structure using Python. Through hands-on exercises involving Breadth-First Search (BFS) and Depth-First Search (DFS), we explored how different traversal strategies affect graph exploration, efficiency, and use cases. The comparison between adjacency list and adjacency matrix representations highlighted the trade-offs between memory usage and access speed, emphasizing that adjacency lists are more efficient for sparse graphs. Additionally, analyzing undirected and directed graphs deepened our understanding of real-world applications such as social networks and network routing. Overall, this activity enhanced our comprehension of graph theory fundamentals and their importance in solving complex computational and real-world problems.

References

[1] S. "Difference between BFS and DFS," *GeeksforGeeks*, 11 Jul. 2025. [Online]. Available: https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/. [Accessed: 18-Oct-2025].

[2]"Comparison between Adjacency List and Adjacency Matrix representation of Graph," *GeeksforGeeks*, 15 Jul 2025. [Online]. Available: https://www.geeksforgeeks.org/dsa/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/. [Accessed: 18-Oct-2025].