UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Directo, Hannah Thea B.

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.

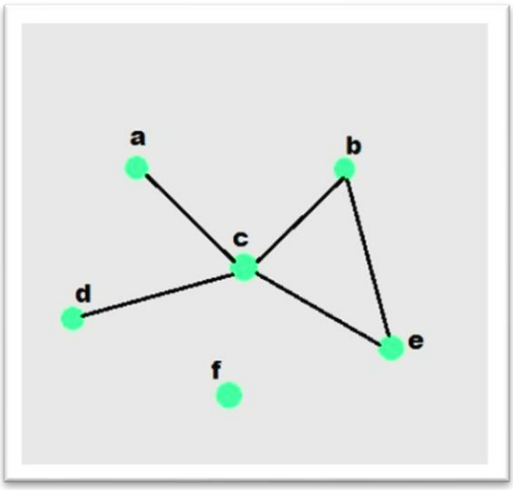

Figure 1. Sample graph with vertices and edges

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

1. When the program runs, it first creates an undirected graph with edges between nodes 0 to 4. The BFS traversal from node 0 visits the nodes in the order [0, 1, 2, 3, 4], and DFS gives the same result. After adding two more edges (4–5 and 1–4), the BFS output changes to [0, 1, 2, 4, 3, 5], while DFS becomes [0, 1, 2, 3, 4, 5].

This shows that BFS explores nodes level by level, while DFS goes deeper before backtracking.

```
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 Screenshot of the Output

2. Breadth-First Search (BFS) and Depth-First Search (DFS) differ in how they explore a graph. BFS uses a queue to visit nodes level by level, making it useful for finding the shortest path in an unweighted graph. DFS uses recursion or a stack to follow one path as far as possible before returning, which is helpful for exploring or analyzing the graph's structure.

   Both methods have a time complexity of (O(V+E)), which depends on the number of nodes and edges. BFS may use more memory because it stores many nodes at each level, while DFS can reach recursion limits in very large graphs. In general, BFS is better for finding the shortest route between nodes, while DFS is more suited for searching and exploring deeper parts of the graph.

3. The code represents the graph using an adjacency list, storing each node with a list of its neighbors in a dictionary. This method is memory-efficient for sparse graphs, as it records only existing edges. Compared to adjacency matrices, which use more memory but allow quicker edge lookups, adjacency lists offer a good balance of space and performance. While edge lists are simpler, they are less efficient for adjacency queries. Thus, adjacency lists are commonly preferred in practical applications involving large, sparse graphs.

4. The current implementation uses an undirected graph, where each edge is bidirectional. When an edge is added between u and v, both directions are stored, creating a two-way connection. To support directed graphs, the add_edge method should be modified to add only v to u's adjacency list, making the edge one-way. BFS and DFS will still function but will follow only the direction of edges, which may affect reachability. Directed graphs are more suitable for applications such as task scheduling and web link structures, where relationships are not reciprocal.

5. Two real-world problems modeled by graphs are supply chain logistics and electric power grids. In supply chains, warehouses and distribution centers are nodes, and routes between them are weighted edges representing cost or delivery time. The graph code must be extended to handle weighted edges and include algorithms like Dijkstra's to find the most efficient routes. BFS and DFS help explore the network and check connectivity. In electric power grids, power plants and substations are nodes, and transmission lines are directed, weighted edges showing capacity or load. The graph should support directed, weighted edges, and BFS and DFS help analyze connectivity and detect faults. Dijkstra's algorithm is needed to optimize power flow. These changes make the graph suitable for solving such complex problems effectively.

IV. Conclusion

This laboratory activity taught us how to build and use graphs in Python. We learned how to represent graphs with lists of connections and how to explore them using BFS and DFS, which visit nodes in different ways. We also saw that to solve real-world problems like supply chains and power grids, graphs need to handle directions and weights on edges. This activity helped us understand the basics of graphs and why they are important for solving many problems in computer science.