UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Regondola Jezreel P.

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.

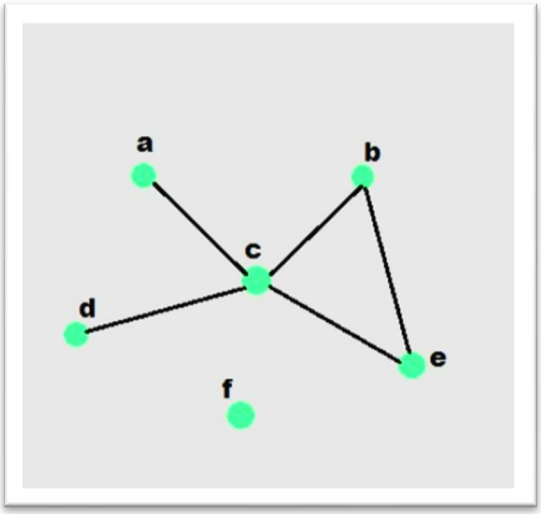

Figure 1. Sample graph with vertices and edges

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.

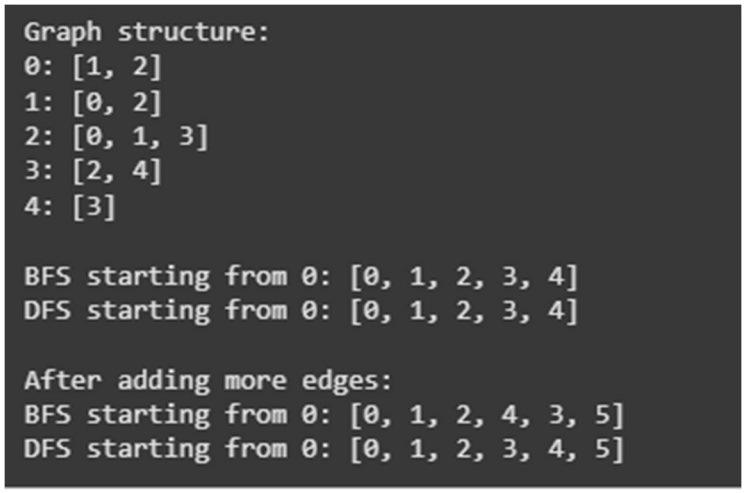

Figure 1 Output of the program

2. BFS uses a queue and checks all nearby nodes first before going deeper, while DFS uses recursion and goes as far as it can in one direction before backtracking. BFS is good for finding the shortest path, while DFS is better for exploring or searching for something deep inside the graph.

3. The program used an adjacency list to represent the graph. This is a simple and memory-saving way to show the connections because it only stores existing edges. On the other hand, an adjacency matrix uses a table that takes up more space but makes it faster to check if two nodes are connected. An edge list can also be used, but it's slower when checking for direct connections between nodes.

4. The graph in the program is undirected, which means every connection goes both ways. If node A connects to node B, then node B is also connected to node A. To make it directed, the code needs to be changed so that edges only go one way. This can be done by removing the second line that adds the reverse connection.

5. Graphs can represent many real-world problems. For example, in social networks, each user can be a node and friendships can be the edges. Using BFS can help find the shortest connection path between two users. Another example is transportation systems, where cities can be nodes and routes can be edges. In this case, DFS can help explore all possible paths, while BFS can find the shortest travel route.

IV. Conclusion

This activity helped me learn how to make and use graphs in Python. I understood how BFS and DFS work and how they are different from each other. BFS visits nodes level by level, while DFS goes deep into one path before returning. This activity helped me see how these algorithms are used in real-life examples like maps and social networks. It also gave me a better understanding of how non-linear data structures work

References

[1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.