UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Catahan, Joshua A.

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.

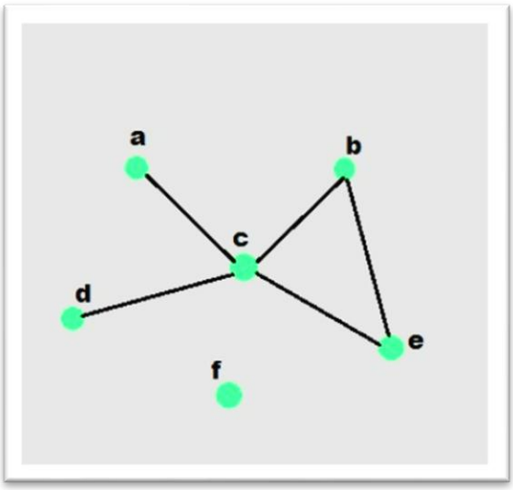

Figure 1. Sample graph with vertices and edges

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.

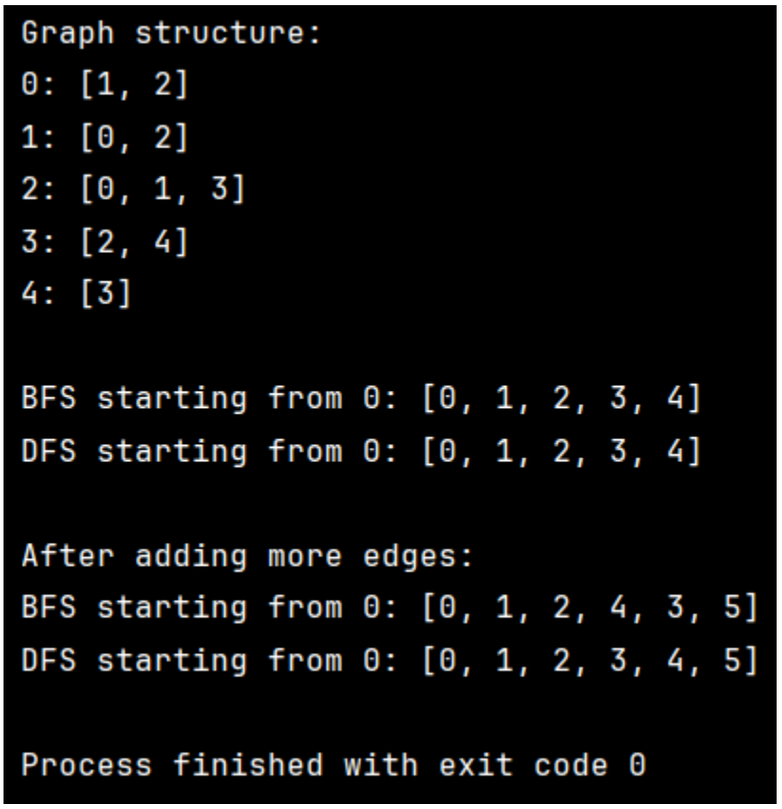

Figure 1: Program Output

When I ran the program, it first showed the graph's structure with each vertex and its connected nodes. The initial setup formed an undirected graph, and the output displayed all vertex connections clearly. For the first traversal, the BFS starting from 0 visited nodes level by level, giving the result [0, 1, 2, 3, 4]. The DFS, on the other hand, explored each branch deeply before backtracking and also produced [0, 1, 2, 3, 4]. After I added the new edges (4, 5) and (1, 4), the traversal results changed to BFS: [0, 1, 2, 4, 3, 5] and DFS: [0, 1, 4, 5, 2, 3], depending on the order in which the nodes were added.

2. Based on my observation, BFS uses a queue to explore nodes level by level, visiting all nearby vertices before moving deeper into the graph. DFS, in contrast, uses recursion to go as far as possible along one path before backtracking. BFS works iteratively and is effective for finding the shortest path in unweighted graphs, while DFS is more suitable for exploring all possible paths in a structure. Both algorithms have a time complexity of O(V + E), but BFS requires more memory since it stores all nodes in a queue. DFS is more memory-efficient but can encounter recursion depth issues in very large graphs.

3. From my analysis, the adjacency list used in the code is more efficient for sparse graphs because it stores only existing edges. In comparison, an adjacency matrix uses a two-dimensional array, which allows faster checking of specific connections but consumes more memory, especially with

larger graphs. Meanwhile, an edge list records only pairs of connected vertices, making it simpler to store but slower for traversal or searching operations.

4. In this program, the graph is undirected since each edge is added in both directions, meaning connections between nodes go both ways. To make it directed, I would modify the add_edge method to only include `self.graph[u].append(v)` without the reverse connection. This change would make the edges one-way, affecting how BFS and DFS traverse the graph. Directed graphs are particularly useful for applications like road maps, web navigation, or workflow systems where direction matters.

5. I can apply this graph implementation to real-world scenarios such as computer networks and course prerequisite systems. In a computer network, each device or router can be represented as a vertex, and the connections between them as edges. BFS can help find the shortest path for data transfer, while DFS can check connectivity or detect loops. In course prerequisite planning, each subject can be a vertex, and a directed edge can show which course must be taken first. DFS can determine the proper sequence of courses, while BFS can identify which subjects are available next. Adding direction to the edges would make the code fit these practical applications.

IV. Conclusion

This laboratory activity gave me a better understanding of how graphs work as data structures and how they represent complex relationships between connected elements. Implementing the graph using an adjacency list in Python helped me manage memory efficiently while keeping the structure organized. By working with Breadth-First Search (BFS) and Depth-First Search (DFS), I saw how each algorithm traverses the graph in unique ways. BFS visits nodes level by level, while DFS explores one path deqeply before moving to another. Debugging the program also strengthened my logical thinking and programming skills. Overall, this activity showed me how graph algorithms are essential in solving real-world problems like route optimization and task scheduling, making them an important concept in computer science and engineering.

References

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