UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Bron, Jhustine 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.

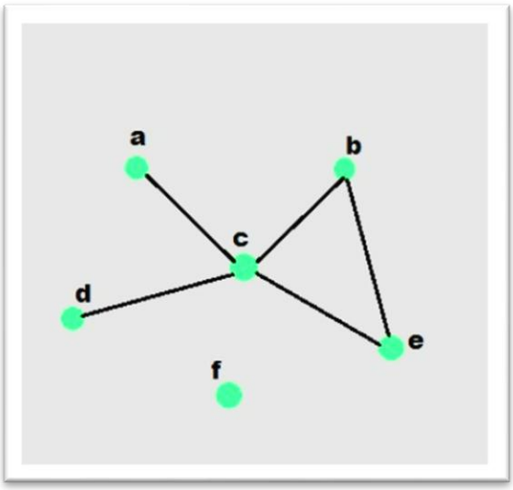

Figure 1. Sample graph with vertices and edges

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. Output of the program:

```
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]
```

2. The main difference between BFS and DFS in this code is how they explore the graph. BFS uses a queue, which means it visits nodes level by level. It is an iterative approach. DFS uses recursion and goes deep into one path before backtracking. That is why DFS is sometimes better for exploring structures like trees or detecting cycles, while BFS is better for finding the shortest path in an unweighted graph.

- BFS → uses a queue, explores level by level
- DFS → uses recursion, explores as deep as possible first
- Both have time complexity O(V + E)
- BFS may use more memory in wide graphs, DFS may use more memory in deep ones

3. The code uses an adjacency list, which is memory efficient since it only stores actual connections. This is especially good for large and sparse graphs. In comparison, an adjacency matrix stores all possible connections in a V x V table, which uses more memory but makes checking if two nodes are connected very fast. An edge list is just a list of all edges and is simple, but slower when checking neighbors.

- Adjacency list → efficient for sparse graphs (used here)
- Adjacency matrix → faster for connection checks but takes more memory
- Edge list → easiest format but not ideal for traversal

4. The graph is undirected, meaning when you add an edge between u and v, the code adds both u → v and v → u. This means connections work both ways automatically. To make it a directed graph, you would remove the line that adds the

reverse connection, making edges one-way only. BFS and DFS would still work, but traversal order could change depending on arrow directions.

DIRECTED:

```python
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)  # I removed this part making it directed

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

The rest of the program is still the same.

If the graph is changed to directed, only one direction of the edge would be stored, which means traversal results will now depend on arrow direction and some nodes may not be reachable unless the edges explicitly allow it. This also introduces the idea of in-degree and out-degree, since each node may now have different numbers of incoming and outgoing edges.

5. One real-world program I could build using this graph implementation is a campus navigation system. Every building or classroom would be a node, and the walkable paths between them would be the edges. Using BFS, the program could find the shortest walking path from one building to another, which would be very useful for new students. To make this practical, I would modify the code to include edge weights for distance or time.

Another real-world application is a task dependency manager**.** Each task would be a node, and an edge from Task A to Task B means “you must finish A before B.” This is a directed graph, so I would modify the add_edge method to only store one direction. DFS would be useful here to detect cycles (which indicate impossible schedules), and I could also implement topological sorting to automatically generate a valid order of execution for all tasks. That would be an actual usable program for project planning or scheduling.

IV. Conclusion

In this task, I explored how graphs work in Python, including how BFS and DFS operate differently, how graphs can be represented using adjacency lists, and how the behavior changes between undirected and directed graphs. I also connected everything to real-world applications by explaining how this code could be used for systems like campus navigation and task management. Overall, this activity helped me understand not just the code itself, but how graph theory is actually applied in practical and useful programs.

References

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

GeeksforGeeks, "Difference between BFS and DFS," *GeeksforGeeks*, Jul. 11, 2025. https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

"Graph Data Structure." https://www.programiz.com/dsa/graph