UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Dispo,Lei Andrew T.

Instructor:
Engr. Maria Rizette H. Sayo

10, 25, 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.

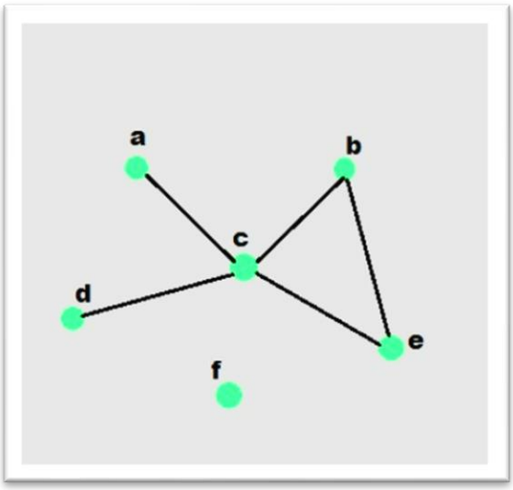

Figure 1. Sample graph with vertices and edges

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. 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, 3, 4, 5]
   DFS starting from 0: [0, 1, 2, 3, 4, 5]

2. Breadth-First Search (BFS) uses a queue data structure to explore nodes level by level. It visits all the neighbors of a node before moving to the next level, making it suitable for finding the shortest path in unweighted graphs. Depth-First Search (DFS), on the other hand, uses recursion (or a stack) to explore as deeply as possible along one branch before backtracking.

   Both BFS and DFS have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. BFS is iterative and ensures that the shallowest nodes are processed first, while DFS's recursive nature explores deeper paths first.

   The advantage of BFS is that it finds the shortest path quickly, while DFS is memory-efficient for sparse graphs and useful for detecting cycles or exploring connected components. However, BFS can consume more memory, and DFS can lead to stack overflow in deep recursions.

3. The adjacency list representation is memory-efficient for sparse graphs since it stores only existing edges. It allows fast traversal of a node's neighbors. In contrast, an adjacency matrix uses a two-dimensional array regardless of edge count, making it suitable for dense graphs but wasteful for sparse ones. An edge list stores all edges as vertex pairs, which is easy to implement but inefficient for checking adjacency or traversing neighbors.

4. The graph in the code is undirected, meaning every edge goes both ways. For example, if there is a connection from node 1 to node 2, there is also one from node 2 to node 1. That's why in the add_edge method, both directions are added. If we are to modify the code here's what we would do:

   ```
   def add_edge(self, u, v):
       if u not in self.graph:
           self.graph[u] = []
       self.graph[u].append(v)
   ```

5. Two real-world problems that can be modeled using graphs are social networks and navigation systems. In a social network, users are nodes and friendships are edges; BFS can find the shortest connection between people, while DFS can explore groups or communities. In navigation, cities or roads are nodes and edges; BFS helps find the shortest path, and DFS can check all possible routes. To improve the code, we can add weights to edges and use algorithms like Dijkstra's or A* for more accurate pathfinding.

IV. Conclusion

In this activity, the implementation of graphs using Python helped demonstrate how nodes and edges represent relationships between data. Through BFS and DFS, we explored different ways to traverse graphs efficiently. The exercise showed how graphs can model real-world systems like social networks and navigation routes, highlighting their importance in computer science and problem-solving.

References

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