UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Ruperto, April Anne 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.

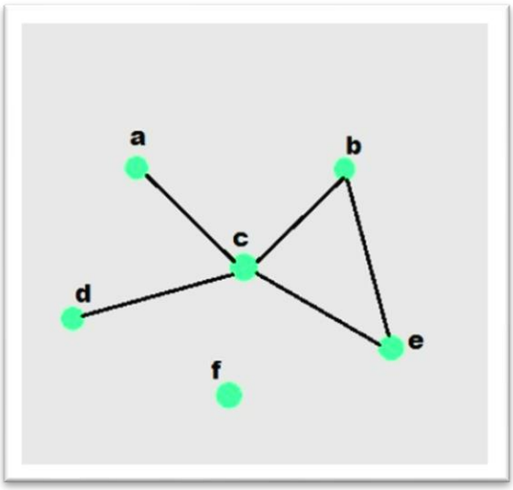

Figure 1. Sample graph with vertices and edges

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

Please follow this link: Ruperto-April-Anne/CPE-201L-DSA-2-A/ /DSA_Lab11.ipynb

1. Output:

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. Breadth-first search (BFS) uses queue to explore nodes level by level. This data structure starts from the source then explores all immediate nodes and so on. The BFS guarantees to find the shortest path in an unweighted graph, but it requires higher memory usage since it holds all neighboring nodes at the same level. Depth-first search (DFS) uses recursion/stack to keep track of its transversal path. It goes as far as possible down one branch, then explores alternative branches. Due to recursion depths limits, it uses stack overflow for deep and large graphs.

3. The provided graph uses adjacency list which is efficient for sparse graphs. The edges in the graph using adjacency list are easier to add or remove and its memory efficient when the graph isn't dense. Adjacency Matric is represented as a 2D array (u x v), adding and removing vertices are costly and its highly inefficient for sparse graph since there's a lot of unused space. While edge list is represented as list of pairs or tuples. It is good for algorithms that process edges directly, but it is slower in finding neighboring nodes and not ideal for transversal-based algorithms.

4. To support a directed graph, we need to remove the second append line *'self.graph[v].append(u)'* which affects the transversal and behavior. The implemented graph is symmetrical with transverse algorithms and by removing the second append line, it can only follow the direction of edges (from u --> v) and the graph can have an unreachable node from a certain point.

5. It can for social networks connections where each represents a person, it can be used for finding the shortest connection between two people using BFS. It can also be used in websites where each node is a website, and each hyperlink is a direct edge. We can use DFS to explore deeply linked pages efficiently.

IV. Conclusion

In conclusion, the graph implementation given here is an excellent way to show how the graph can be represented and how the graph traversal through BFS and DFS can be done. The time complexity of both traversal operations is the same, i.e., (O(V + E)), however, they differ in the data structures they use and the methods of exploration - BFS is an iterative, queue-based approach which is used for finding shortest paths and DFS employs recursion to drill down to the deep paths.

The use of adjacent list representation offers a space-efficient and a more versatile structure which is especially useful for sparse graphs than adjacent matrices or edge lists. Besides that, the undirected graph implementation makes the connections symmetric, but small changes can turn it into a directed graph which can be used to handle one-way relationships. This implementation can be extended to be used as a solution to real-world problems like social network analysis or web page crawling where BFS and DFS are the main instruments for navigating and analyzing complex networks.

References

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