UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Talagtag Mark Angel T.

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.

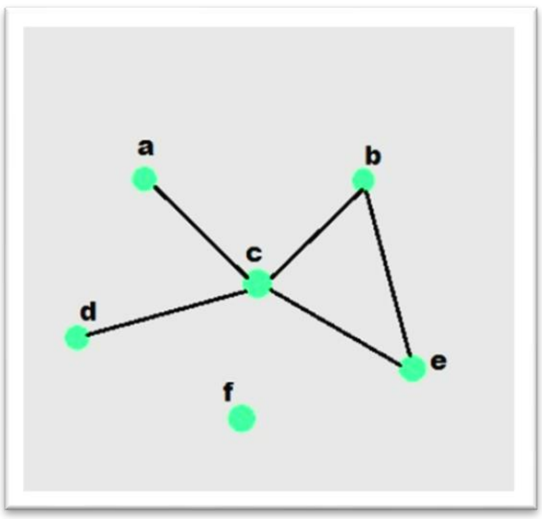


Figure 1. Sample graph with vertices and edges

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
```

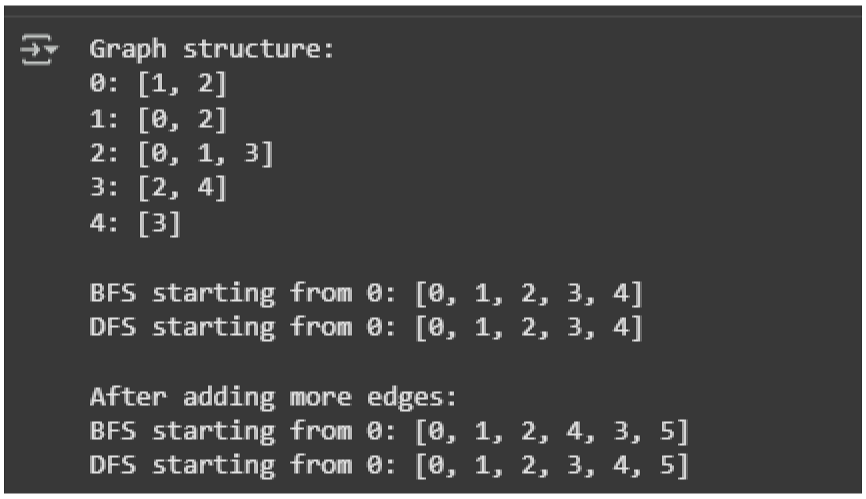
```
g = Graph()

# 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.



```
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.

BFS uses a **queue** for iterative, level-by-level traversal, while DFS uses **recursion** to explore paths deeply before backtracking. Both have a time complexity of $O(V+E)$, but DFS can risk stack overflow on deep graphs. BFS is better for shortest paths, while DFS is useful for exploring all reachable nodes.

3. An adjacency list is space-efficient for sparse graphs and allows fast neighbor lookups. An adjacency matrix uses more memory but supports constant-time edge checks, ideal for dense graphs. An edge list is simple but slower for neighbor queries.

4.
The graph is undirected because each edge is stored in both directions, making traversal bidirectional. To support directed graphs, only add edges in one direction during add_edge. This change affects traversal, making pathfinding follow edge direction, suitable for applications like web links or dependency graphs.

5.
For a social network, BFS can find the shortest connection between users, while DFS explores friend groups. For web crawling, BFS explores pages level by level, and DFS dives deep into links. Additional algorithms like Dijkstra or PageRank may be added for weighted paths or ranking.

IV. Conclusion

V. In summary, the decision between BFS and DFS is based on the particular traversal requirements, with BFS being appropriate for shortest path applications and DFS suitable for probing deep in the graph, both using various data structures and techniques. The adjacency list implementation offers a time-efficient and versatile means of representing graphs, particularly sparse ones, with others such as adjacency matrices and edge lists used in other applications. Lastly, converting the graph from undirected to directed impacts edge management and traversal behavior, allowing modeling of various real-world issues like social networks and web crawling, for which BFS and DFS are central along with other algorithms.

References

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

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. “*Introduction to Algorithms*,” MIT Press, 2009.

[3] Sedgewick, R., & Wayne, K. “*Algorithms*,” Addison-Wesley, 2011.

[4] GeeksforGeeks. “Graph Data Structure and Algorithms,” GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

[5] MIT OpenCourseWare. “Graph Algorithms,” MIT, 2024. [Online]. Available: https://ocw.mit.edu

[6] Python Software Foundation. “collections.deque — Container datatypes,” Python Docs, 2024. [Online]. Available: https://docs.python.org/3/library/collections.html#collections.deque