UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Villacin, Justine R.

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.

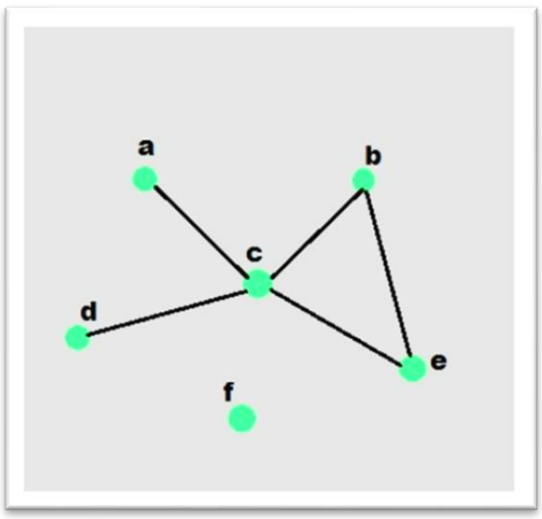


Figure 1. Sample graph with vertices and edges

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)}")
```

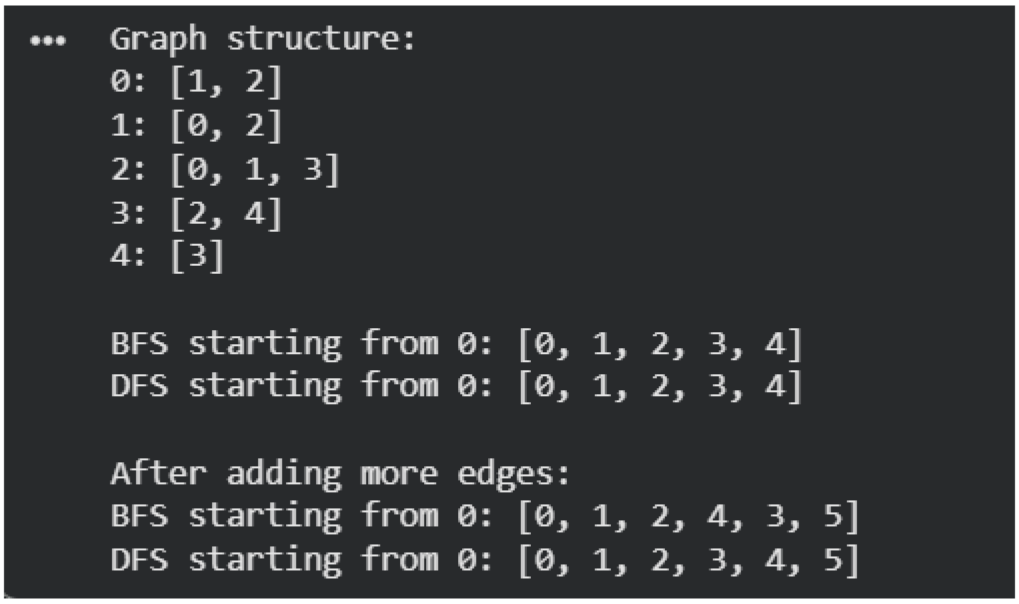
Questions:

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

Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals, books, magazines, reports, or web articles please cite and reference them using the IEEE format.

Please take note of the styles on the style ribbon as these would serve as the style format of this laboratory report. The body style is Times New Roman size 12, line spacing: 1.5. Body text should be in Justified alignment, while captions should be center-aligned. Images should be readable and include captions. Please refer to the sample below:



```
•••  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]
```

Figure 1 Output

If an image is taken from another literature or intellectual property, please cite them accordingly in the caption. Always keep in mind the Honor Code [1] of our course to prevent failure due to academic dishonesty.

ANSWERS:

1. Code Output

The code will first show the graph's connections, which are: node 0 is connected to 1 and 2, node 1 is connected to 0, 2, and 4, and so on. The BFS traversal starting from node 0 will visit nodes in the order [0, 1, 2, 4, 3, 5], exploring all neighbors first before moving deeper. The DFS traversal from the same node will visit [0, 1, 2, 3, 4, 5], going as far down a path as possible before backtracking.

2. BFS vs. DFS Differences

The main difference is that BFS uses a queue to explore a graph level-by-level, which is great for finding the shortest path but uses more memory. In contrast, DFS uses a stack (or recursion) to go deep down one branch first, which uses less memory but does not guarantee the shortest path. The recursive DFS is simpler to write but can sometimes cause errors on very large graphs, while the iterative BFS is more predictable.

3. Graph Representation Comparison

The provided code uses an adjacency list, which is efficient in terms of memory and is ideal for graphs that don't have many connections. An alternative is an adjacency matrix, which is a table that quickly tells you if two nodes are connected but uses much more memory, making it better only for graphs that are very densely connected.

4. Directed vs. Undirected Graphs

The current graph is undirected, meaning an edge between two nodes goes both ways, which is why the add_edge method adds each node to the other's list. To make it a directed graph, you would only add the connection in one direction, which would change the traversals to only follow the direction of the arrows. This is useful for modeling one-way relationships like web links or task dependencies.

5. Real-World Problems

Two real-world problems are social network friend suggestions and network packet routing. For friend suggestions, you could use BFS to find "friends-of-friends." For finding a routing path in a network, BFS would efficiently find the shortest path. To fully solve these, you might need to add features like edge weights and implement algorithms like Dijkstra's for more complex routing.

IV. Conclusion

In short, this graph code is a versatile tool where BFS is best for shortest paths and DFS for deep exploration, and with small changes, it can be adapted for many different real-world uses.

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. (2022). *Introduction to algorithms* (4th ed.). The MIT Press. https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

[3] GeeksforGeeks. (2023). *Graph Data Structure And Algorithms*. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/