UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Maringal, Czer Justine D.

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.

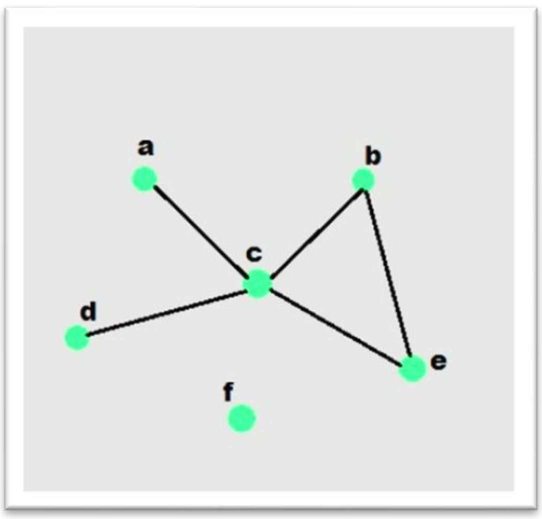


Figure 1. Sample graph with vertices and edges

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
```

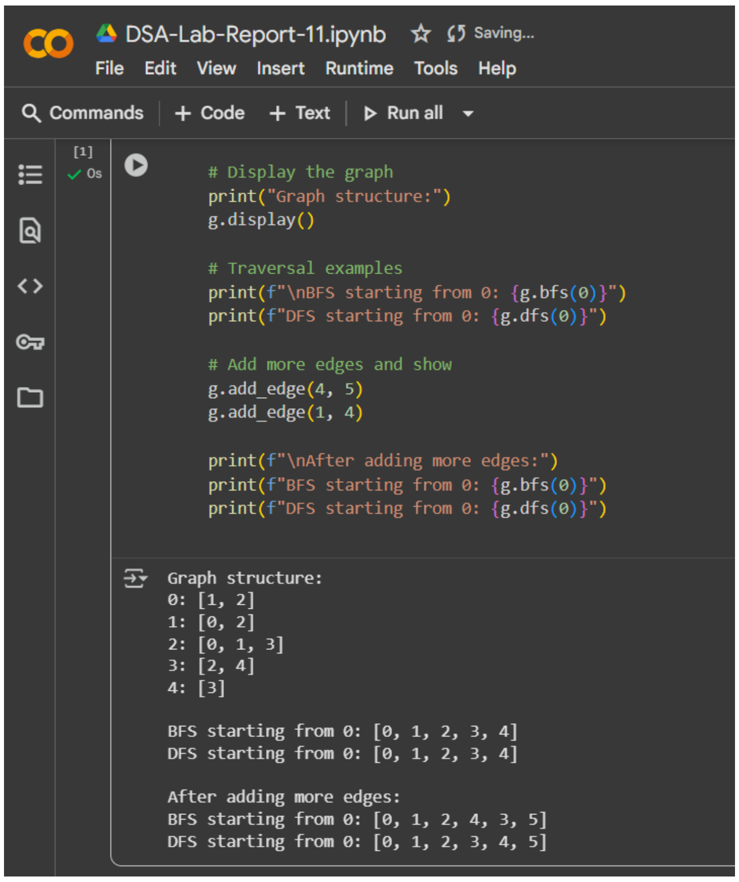
```
g = Graph()

# 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. The output looks like this:


DSA-Lab-Report-11.ipynb
Saving...
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[1]
0s

```
# 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)}")
```

```
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]
```

After adding more edges the graph looks like this

0: [1, 2]

1: [0, 2, 4]

2: [0, 1, 3]

3: [2, 4]

4: [3, 5, 1]

5: [4]

2. BFS (Breadth-First Search) uses a queue and checks each level one by one, while DFS (Depth-First Search) uses recursion and goes as deep as possible before going back. BFS is good for finding the shortest path, and DFS is good for exploring all paths. By that, BFS can use more memory while DFS can get an errors if the graph is too deep.

3. In this code, the adjacency list approach to stores each vertex with its connected nodes. This is efficient and simple, especially for sparse graphs. An adjacency matrix uses more memory but allows faster edge lookups, while an edge list is simpler but slower for traversal. The adjacency list is best for BFS and DFS since it saves space and makes node access easy.

4. The graph is undirected, meaning edges connect both ways (1-2 and 2-1). This is suitable for mutual relationships like friendships. To make it directed, we just remove the line self.graph[v].append(u) so edges go only one way. Traversals like BFS and DFS will still work

but will follow directions, which is useful for modeling one way systems like web links or task orders.

5. In social networks, BFS can find the shortest connection between users. In web crawling, DFS can explore pages deeply before backtracking. To apply this code, I'm going to add features like edge weights or pathfinding algorithms such as Dijkstra's or A* to handle more complex situations.

IV. Conclusion

In this activity, I learned how graphs work in Python using BFS and DFS. I understood how BFS explores nodes level by level using a queue, while DFS goes deep using recursion. I also learned about different graph representations and how undirected and directed graphs differ. Overall, this activity helped me understand how graph traversal works and how it can be used in real-life applications like social networks and web searches.

References

[1] GeeksforGeeks. (2025, August 28). Breadth first search or BFS for a graph. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/

[2] GeeksforGeeks. (2025a, July 15). Introduction to graph Data Structure. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/introduction-to-graphs-data-structure-and-algorithm-tutorials/

[3] GeeksforGeeks. (2025c, October 3). Reallife Applications of Data Structures and Algorithms (DSA). GeeksforGeeks. https://www.geeksforgeeks.org/dsa/real-time-application-of-data-structures/