UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Palmes, Lewis Clark L.

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.

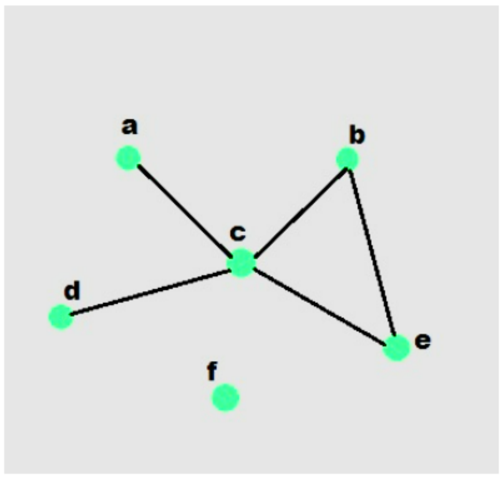


Figure 1. Sample graph with vertices and edges

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.

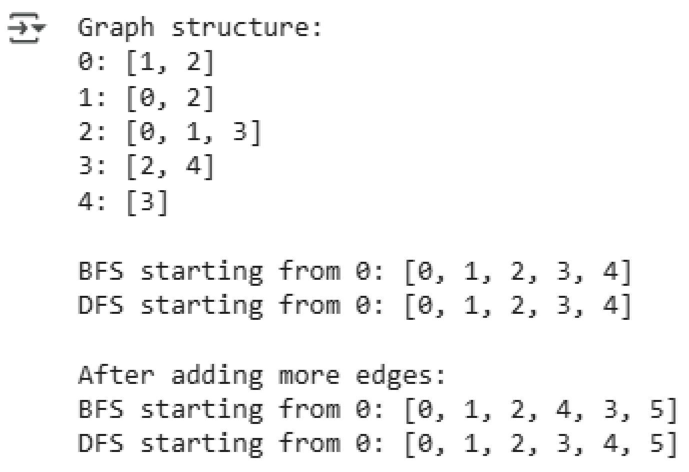
4. The graph in the code is implemented as undirected. Analyze the implications of this design choice on the add_edge method and the overall graph structure. How would you modify the code to support directed graphs? Discuss the changes needed in edge addition, traversal algorithms, and how these modifications 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. What will be the output of the following codes?

The output of the provided code would be as follows:



```
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 Screenshot of the output program

The graph is initially represented as undirected, and both BFS and DFS visit all reachable nodes from node 0, showing different traversal orders based on their respective strategies. The addition of more edges doesn't change the BFS/DFS outcome significantly except for BFS's order, where it explores the newly added node 5.

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?

ANSWER:

BFS uses a queue to explore nodes level by level, ensuring the shortest path in unweighted graphs. DFS uses recursion/stack, exploring deep into a branch before backtracking. BFS is iterative and uses more memory, while DFS is recursive but prone to stack overflow in deep graphs. Both have time complexity O(V+E)

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.

   ANSWER:

   The adjacency list is space-efficient for sparse graphs, while an adjacency matrix uses V×V space, making it inefficient for sparse graphs. Edge lists are simple but not efficient for neighbor lookups. The adjacency list is best for most real-world scenarios.

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.

   ANSWER:

   In undirected graphs, edges are bidirectional. For directed graphs, modify add_edge to only add u→v, not v→u. This affects traversal, which would follow directed edges only.

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.

   ANSWER:

   For shortest path (city roads), BFS is useful in unweighted graphs. For web scraping, DFS helps in recursively following links. For both, modifications like weighted graphs or cycle detection might be needed.

IV. Conclusion

This laboratory activity helped us better understand how graphs work and how they can be represented and explored using Python. By building an undirected graph and applying both BFS and DFS, we saw how different traversal methods affect the way we explore connections between nodes. Working with an adjacency list made it easier to manage and visualize the graph, especially for larger or more complex structures. Overall, this exercise not only improved our coding skills but also showed how useful graphs are in real-life situations like navigation, web crawling, and social networks

References

[1] “Google Colab.”
https://colab.research.google.com/drive/1dwXu7mBFnFpiWC3lIIkvoym0_-OgMYdy#scrollTo=6osaM4hBT-8C

[2] “Graph Traversal | Compile n run,” *Compile N Run*.
https://www.compilenrun.com/docs/fundamental/algorithm/graph-algorithms/graph-traversal/

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

[4] GeeksforGeeks, “Adjacency list representation,” *GeeksforGeeks*, Jul. 23, 2025.
https://www.geeksforgeeks.org/dsa/adjacency-list-meaning-definition-in-dsa/

[5] GeeksforGeeks, “Adjacency matrix representation,” *GeeksforGeeks*, Mar. 19, 2025.
https://www.geeksforgeeks.org/dsa/adjacency-matrix/

[6] GeeksforGeeks, “Why is the complexity of both BFS and DFS O(V+E)?,” *GeeksforGeeks*, Jul. 23, 2025.
https://www.geeksforgeeks.org/why-is-the-complexity-of-both-bfs-and-dfs-ove/

[7] Greg Hogg, “4 ways to represent graph data structures - edge list, adjacency list / matrix, object & pointer,” *YouTube*. Mar. 02, 2024. [Online]. Available: https://www.youtube.com/watch?v=J_C3sUVH4ZE

[8] Lewis-Clark-Palmes, “CPE-201L-DSA-2-A/LAB-11.ipynb at main · Lewis-Clark-Palmes/CPE-201L-DSA-2-A,” *GitHub*.
https://github.com/Lewis-Clark-Palmes/CPE-201L-DSA-2-A/blob/main/LAB-11.ipynb