



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Disomnong, Jalilah M.

Instructor:
Engr. Maria Rizette H. Sayo

10/18/2025

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.

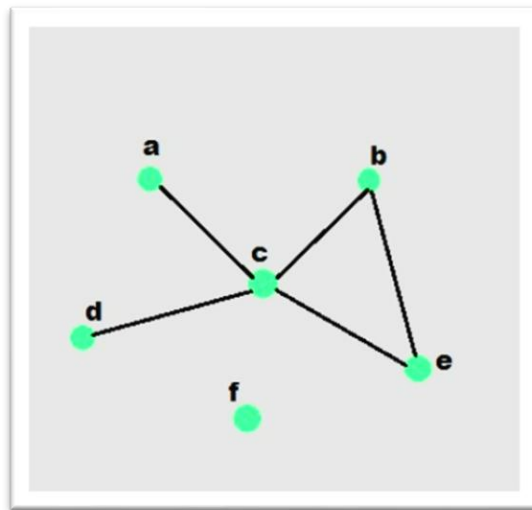


Figure 1. Sample graph with vertices and edges

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"\nDFS starting from 0: {g.dfs(0)}")

# Add more edges and show
g.add_edge(4, 5)
g.add_edge(1, 4)

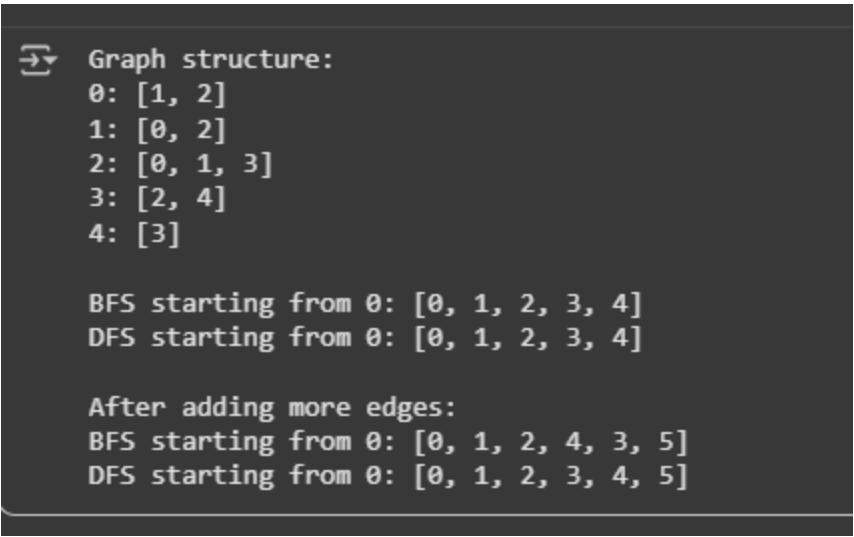
print(f"\nAfter adding more edges:")
print(f"BFS starting from 0: {g.bfs(0)}")
print(f"DFS starting from 0: {g.dfs(0)}")

```

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

A screenshot of a terminal window with a dark background and light-colored text. It displays the output of a graph program. The text is 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 link: [Lab 11 - Colab](#)

In question number 1, the program creates an undirected graph where each vertex is connected to others through edges. When the Breadth-First Search (BFS) starts from vertex 0, it visits nearby vertices first before moving farther, resulting in the output [0, 1, 2, 3, 4]. The Depth-First Search (DFS), on the other hand, goes deeper into one path before backtracking, but in this case, it also produces [0, 1, 2, 3, 4] since the graph is simple and connected. After adding new edges (4–5) and (1–4), the BFS output becomes [0, 1, 2, 4, 3, 5], while DFS becomes [0, 1, 2, 3, 4, 5].

In question number 2, Breadth-First Search (BFS) is a graph traversal method that explores all nodes at the same level before moving to the next level. Depth-First Search (DFS), on the other hand, starts from the root and continues deeper along one path until it reaches a node with no unvisited neighbors. BFS uses a queue to find the shortest path and is best for locating vertices close to the source. Meanwhile, DFS uses a stack and is more effective when the solution lies farther from the starting point (GeeksforGeeks, 2025).

In question number 3, the program uses an adjacency list, which stores each vertex and its connected nodes. It is simple, saves memory, and works well for graphs with fewer connections. An adjacency matrix uses a table to show all possible connections, making it faster to check if an edge exists but uses more space. An edge list only lists all the edges, making it easy to build but slower to search.

In question number 4, the graph in the code is undirected, meaning each edge connects two vertices in both directions. When an edge is added between u and v , both can reach each other. To make the graph directed, the `add_edge` method should only add one direction by removing `self.graph[v].append(u)`. BFS and DFS will still work but will follow one-way paths. This change is useful for showing one-way systems like traffic routes or website links.

In question number 5, BFS and DFS are both useful in many real-world problems. BFS is often used for tasks like finding the shortest path or working with bipartite graphs, especially when all edges have the same weight. DFS, on the other hand, is helpful in exploring acyclic graphs, finding strongly connected components, and similar tasks. Both algorithms can also be applied in problems such as topological sorting and cycle detection, depending on what the graph needs to represent.

IV. Conclusion

In this laboratory activity, I learned about nonlinear data structures, particularly graphs, and how they represent relationships between data using vertices and edges. By implementing graphs in Python, I was able to understand how connections work and how to traverse them using BFS and DFS. I discovered that unlike linear structures such as arrays or linked lists, graphs can connect data in multiple directions, making them useful for real-world problems like route finding or network mapping.

References

- [1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.
- [2] Google. (2024). Google Colaboratory. Retrieved April 18, 2024, from <https://colab.research.google.com/>
- [3] GeeksforGeeks. (2025, July 11). Difference between BFS and DFS. Retrieved from <https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/>
- [4] GeeksforGeeks. (2025b, July 23). What is Undirected Graph? | Undirected Graph meaning. Retrieved from <https://www.geeksforgeeks.org/dsa/what-is-undirected-graph-undirected-graph-meaning/>