UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

# Implementation of Graphs

*Submitted by:*
Delinia, Filjohn B.

*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.

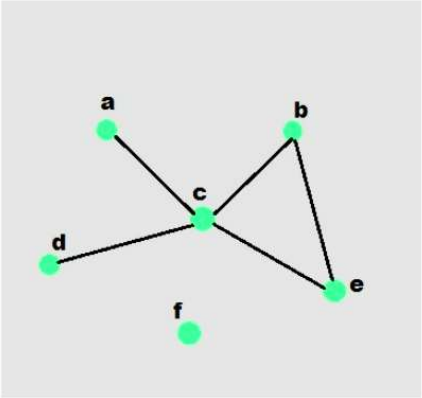


Figure 1. Sample graph with vertices and edges

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. When the given graph code is executed, it first constructs an undirected graph with the edges 0–1, 0–2, 1–2, 2–3, and 3–4. The graph structure at this stage shows that vertex 0 is connected to 1 and 2, vertex 1 is connected to 0 and 2, vertex 2 connects to 0, 1, and 3, vertex 3 connects to 2 and 4, and vertex 4 connects only to 3. Performing a Breadth-First Search (BFS) starting from vertex 0 results in the traversal order [0, 1, 2, 3, 4], while a Depth-First Search (DFS) from the same starting vertex also produces [0, 1, 2, 3, 4]. After additional edges 4–5 and 1–4 are added, the BFS traversal becomes [0, 1, 2, 4, 3, 5], while the DFS traversal results in [0, 1, 2, 3, 4, 5].

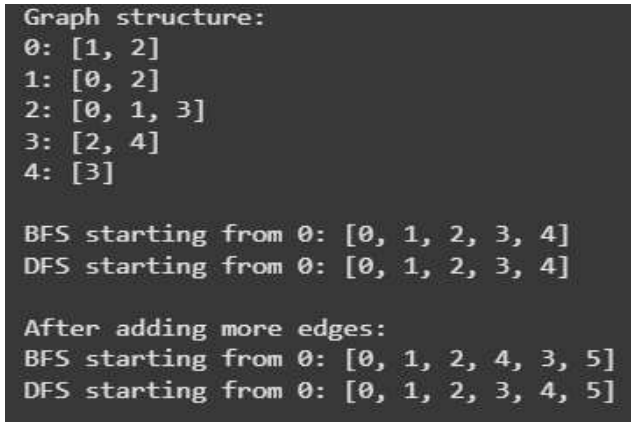


Figure 1.

2. The main difference between the BFS and DFS implementations lies in their traversal strategies and the data structures they employ. BFS uses a queue, specifically a **deque** in Python, to explore the graph level by level, visiting all neighbors of a vertex before moving deeper. It follows an iterative approach, making it efficient for finding the shortest path in unweighted graphs but at the cost of higher memory usage since it stores all neighbors in the queue. DFS, on the other hand, uses recursion, which internally relies on a stack to explore as far as possible along each branch before backtracking. This approach is more memory-efficient for sparse graphs and is often simpler to implement, but it can lead to deep recursion that may cause stack overflow in very large graphs. Both algorithms have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges, but their traversal orders differ significantly.

3. The graph in the code uses an adjacency list representation implemented with a Python dictionary. This approach stores each vertex as a key, with its connected vertices in a list. Compared to other representations, such as adjacency matrices and edge lists, the adjacency list is more space-efficient, especially for sparse graphs. An adjacency matrix,

which uses a two-dimensional array, allows O(1) time edge lookup but requires $O(V^2)$ space, making it inefficient for large graphs with few edges. Meanwhile, an edge list is the simplest representation, storing pairs of connected vertices, but it is slower for traversals and edge lookups. Thus, the adjacency list offers the best balance of efficiency and simplicity for general-purpose graph algorithms.

4. The code currently defines an undirected graph, meaning every time an edge is added between two vertices u and v, both **self.graph[u].append(v)** and **self.graph[v].append(u)** are executed. This ensures that traversal works in both directions, which models relationships that are mutual, such as friendships or bidirectional roads. To modify the code to support directed graphs, only one direction of the edge should be stored by removing the line that adds the reverse connection. This change would allow one-way traversal, meaning algorithms like BFS and DFS would follow directed paths only. As a result, some vertices might become unreachable depending on edge directions, and new algorithms such as topological sorting or strongly connected components could be applied to analyze direction-dependent properties.

5. **Social Networks**
    - In social networks, **each user is represented as a vertex**, and **friendships or connections are represented as edges** between them.
    - **Breadth-First Search (BFS)** can be used to find the **shortest path** between two users — for example, determining how many connections or "degrees of separation" exist between people.
    - **Depth-First Search (DFS)** can explore **entire friend groups** or connected components to identify **communities or clusters of users**.
    - To enhance the provided graph implementation for this purpose, it could be extended to include **weighted edges** that represent the **strength or frequency of interactions** between users.
    - Other useful algorithms could include **Dijkstra's algorithm** for shortest paths or **community detection algorithms** for social group analysis.

**Web Crawlers**

- In a web crawling application, **each webpage is treated as a vertex**, and **hyperlinks are directed edges** connecting one page to another.
- **BFS** allows the crawler to explore **webpages layer by layer**, ensuring all links at a given depth are visited before moving deeper — this helps **control crawl depth** and avoid infinite loops.

- **DFS**, on the other hand, performs a **deep exploration**, following each link chain as far as possible before backtracking, which is useful for **in-depth analysis of specific site branches**.
- To make the current graph code suitable for this scenario, it should support **directed edges**, **cycle detection** to avoid re-crawling pages, and possibly **edge attributes** like URL weights or priorities.
- Additional algorithms such as **PageRank** could also be integrated to **evaluate the importance or popularity of web pages** based on their link structures.

# IV. Conclusion

In summary, the provided graph implementation demonstrates the fundamental principles of graph theory through practical applications of Breadth-First Search (BFS) and Depth-First Search (DFS). Both algorithms highlight different yet complementary ways to explore and analyze relationships between connected data. The use of an adjacency list makes the implementation efficient and adaptable, while the undirected design illustrates mutual connections common in real-world systems. By extending this foundation to include directed or weighted edges, the graph can model more complex problems such as social networks, web crawling, and pathfinding. Overall, this topic emphasizes how graph structures and traversal algorithms form the backbone of many modern technologies, enabling efficient problem-solving in areas like communication, data organization, and network analysis.

# References

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

[2] GeeksforGeeks. (2023, June 16). *Breadth First Search (BFS) and Depth First Search (DFS) for a Graph*. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[3] Kumar, A. (2022, October 10). *Graph representations: Adjacency list and adjacency matrix*. GeeksforGeeks. https://www.geeksforgeeks.org/graph-and-its-representations/

[4] GeeksforGeeks. *Difference between BFS and DFS*. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

[5] Baeldung. (2023, January). *Time and Space Complexity of Adjacency Matrix and List*. Retrieved from https://www.baeldung.com/cs/adjacency-matrix-list-complexity