



UNIVERSITY OF CALOOCAN CITY  
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

---

# Implementation of Graphs

---

*Submitted by:*

Sorellano, John Kenneth T.

*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.

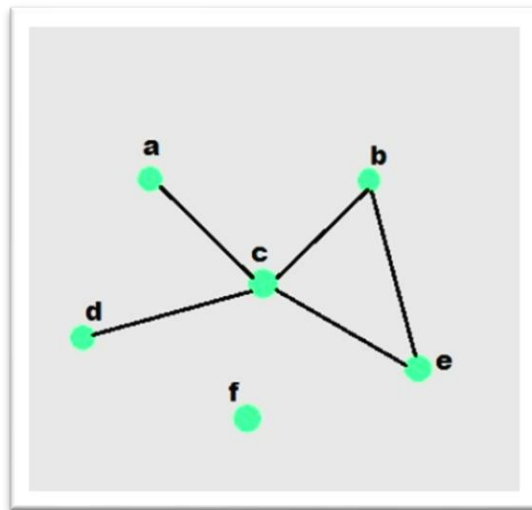


Figure 1. Sample graph with vertices and edges

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"\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"\nDFS 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?

```
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]

Process finished with exit code 0
```

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?
- The BFS are Iterative approach, and used a queue (FIFO), the traversal pattern exploring the elements level by level, in implementation BFS use deque, also it is Better for finding shortest paths in unweighted graphs but use more memory to store nodes at each level. The DFS are in recursive approach, used a stack implementation, the traversal pattern explores as deep as possible, DFS advantage is simple and elegant but at risk of stack overflow for large graphs.
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.
- Adjacency list is best for general-purpose graphs. Adjacency matrix is better when fast edge existence checking is needed. Edge list is good for algorithms.
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.
- Because edges in an undirected graph are bidirectional, `add_edge(u, v)` ensures symmetric connections by adding `v` to `u` as well as `u` to `v`. Reverse addition is removed to accommodate a directed graph, which affects traversal and makes direction-sensitive algorithms possible.

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.
- Graphs can represent web crawling (pages as nodes, hyperlinks as edges) where BFS or DFS investigates connected content, and social networks (people as nodes, friendships as edges) where BFS discovers close relationships. Depending on the application, you would modify the code by adding features like cycle detection, edge weights, or support for directed edges.
  -

## IV. Conclusion

Graphs can represent web crawling (pages as nodes, hyperlinks as edges) where BFS or DFS investigates connected content, and social networks (people as nodes, friendships as edges) where BFS discovers close relationships. Depending on the application, you would modify the code by adding features like cycle detection, edge weights, or support for directed edges.

## References

- Ghadge, A. (n.d.). *BFS and DFS in graphs*. Retrieved October 18, 2025, from <https://ajinkyaghadge.github.io/BFS-and-DFS-in-Graphs/>
- FreeCodeCamp. (n.d.). *Graph algorithms in Python: BFS, DFS, and beyond*. Retrieved October 18, 2025, from <https://www.freecodecamp.org/news/graph-algorithms-in-python-bfs-dfs-and-beyond/>
- GeeksforGeeks. (2025, July 15). *Comparison between adjacency list and adjacency matrix representation of graph*. Retrieved October 18, 2025, from <https://www.geeksforgeeks.org/dsa/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/>
- Gallucci, O. A. (n.d.). *Graph algorithm basics, BFS and DFS*. Retrieved October 18, 2025, from <https://oliviagallucci.com/graph-algorithm-basics-bfs-and-dfs/>