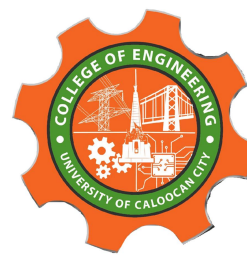




UNIVERSITY OF CALOOCAN CITY  
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

---

# Implementation of Graphs

---

*Submitted by:*  
Adoracion, jerick Dave 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.

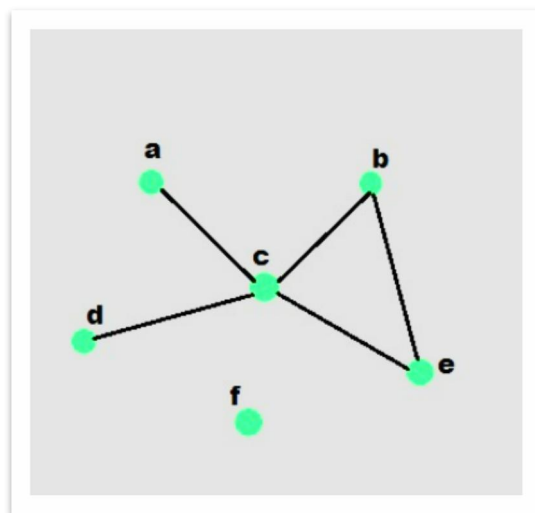


Figure 1. Sample graph with vertices and edges

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

Questions:

1. What will be the output of the following codes?

Graph structure:

0: [1, 2]  
1: [0, 2]  
2: [0, 1,  
3: [2, 4]  
4: [3]

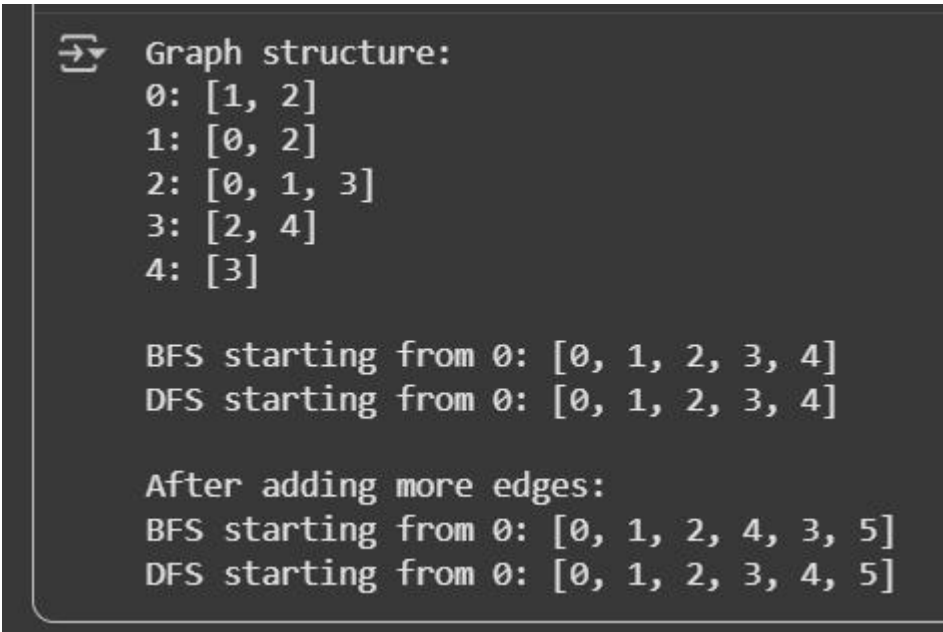
BFS starting from 0: [0, 1, 2, 3, 4]

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]



```
➡ 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: Program Output

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?

In this program, the BFS or **Breadth-First Search**, it used the deque function wherein it follows a certain order of First In, First Out. starting layer by layer from the first node. Through this, nodes are visited in an orderly manner and best for finding the shortest path possible.

Meanwhile, the DFS of **Depth-First Search**, follows a natural stack order of Last In, First Out. This way, where the most recently discovered node is explored next. As a result, DFS dives deep into the graph, following one path as far as it can go before backtracking and exploring alternative routes. This is best for finding the best path, mostly used for puzzles, mazes, and solving.

The **recursive** function of **DFS** explores the graph by following a single path as far as it can go, automatically going back through the stack when it reaches a dead end. Meanwhile, the **iterative** function of **BFS** systematically explores nodes level by level (like a wave) using a queue, ensuring that all nodes at the current distance from the start are processed before moving deeper.

The BFS relies of Breadth first search, which means, it is best for getting the shortest route. Meanwhile, we might encounter errors in deep graphs.

The DFS relies of Depth first search, which means, it is best for getting the best route. Meanwhile, we must need a bigger storage if we will be working on a bigger 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.**

The adjacency list utilized in the given graph implementation achieves a balance between memory efficiency and traversal speed, especially for sparse graphs. Adjacency matrices are more appropriate for dense graphs or situations necessitating rapid edge existence verification, whereas edge lists are optimal for storing sparse graphs, albeit with the drawback of slower neighbor lookup and traversal processes. The selection of representation is significantly influenced by the graph's density, the required operations, and the application's performance considerations.

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.**

Based on the graph implemented in the program, it ensures mutual **connectivity** between vertices. It simplifies traversal and connectivity operations, and it requires storing edges in both adjacency lists, slightly increasing memory usage.

The `add_edge` method must exclusively add edges from the source to the destination to accommodate directed graphs. Traversal algorithms retain their structural integrity, yet their functionality alters, reachability becomes asymmetric, necessitating the Implementation of new algorithms such as topological sorting or cycle detection based on the specific application. This alteration expands the graph's relevance to direction-sensitive issues, yet complicates analysis and traversal.

- 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.**

If I were to play a maze game online, I will use the DFS method, this way, I can determine which path is the best to avoid dangers, and traps. Providing me a better chance of survival. In economics, I can also use DFS to determine which product is best for the market, giving me the edge before my competitors making me attract more customers.

## IV. Conclusion

The conclusion expresses the summary of the whole laboratory report as perceived by the authors of the report.

In this project, I acquired knowledge on the representation and traversal of graphs in Python utilizing BFS and DFS algorithms. Utilizing an adjacency list with a dictionary demonstrated the efficiency of storing and accessing connections between nodes, particularly in sparsely connected graphs. BFS, utilizing its iterative queue methodology, facilitated my comprehension of exploring nodes systematically and determining the shortest path. DFS, using recursion, showed me how to explore deeply along a path and backtrack when needed. Comparing the two methods also taught me about their differences in memory usage and traversal order.

I also explored real-world applications like planning escape routes in a maze game and analyzing the market and its economy. These examples helped me see how graphs can model relationships and connections in practical problems. I learned that sometimes graphs need extra features such as weighted or directed edges to handle more complex scenarios. Overall, this project improved my understanding of graph theory and made me more confident in using algorithms like BFS and DFS in Python.

## References

- [1] **Co Arthur O.** "University of Caloocan City Computer Engineering Department Honor Code," UCC-CpE Departmental Policies, 2020.
- [2] **GeeksforGeeks**, "Breadth First Search or BFS for a Graph," *GeeksforGeeks*, Aug. 28, 2025. [Online]. Available: <https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/>. [Accessed: Oct. 18, 2025].
- [3] **freeCodeCamp**, "Graph Algorithms in Python: BFS, DFS, and Beyond," *freeCodeCamp*, Sep. 3, 2025. [Online]. Available: <https://www.freecodecamp.org/news/graph-algorithms-in-python-bfs-dfs-and-beyond/>. [Accessed: Oct. 18, 2025].
- [4] **Fiveable**, "Applications of BFS and DFS," *Fiveable*, [Online]. Available: <https://fiveable.me/data-structures/unit-11/applications-bfs-dfs/study-guide/tvkrm6qmUv4ZBNDc>. [Accessed: Oct. 18, 2025].
- [5] **Fiveable**, "Breadth-First Search (BFS) Algorithm and Applications," *Fiveable*, [Online]. Available: <https://fiveable.me/data-structures/unit-11/breadth-first-search-bfs-algorithm/study-guide/3yxmer0DnvoLRNV>. [Accessed: Oct. 18, 2025].