



UNIVERSITY OF CALOOCAN CITY  
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

---

# Implementation of Graphs

---

*Submitted by:*  
Villanueva, Bryan O.

*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.

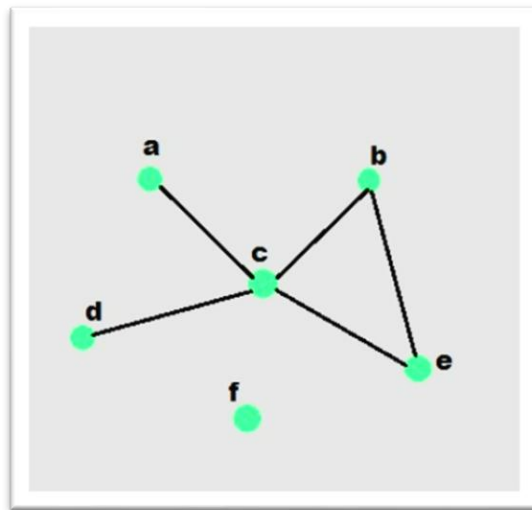


Figure 1. Sample graph with vertices and edges

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

```
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, 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)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)

                for neighbor in self.graph.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)

        return result
```

Figure 1 Input

```
def dfs(self, start):
    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):
    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)
```

Figure 2 Input

```
# 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)}")
```

Figure 3 Input and Output

**Questions:**

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

- After debugging it the output will show

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]

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?

- BFS and DFS work differently even though they both search a graph. BFS uses a queue and works in an iterative way, which means it visits nodes level by level, starting from the closest nodes first. This is good for finding the shortest path but uses more memory. DFS, on the other hand, uses recursion, so it goes deep into one path before coming back to explore others. This approach uses less memory but does not guarantee the shortest path. Both algorithms have the

same time complexity of  $O(V + E)$ , but BFS is better when we need shortest paths while DFS is better for exploring structures deeply.

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 code uses an adjacency list, which is a very memory-efficient way to store a graph because each node only keeps a list of its neighbors. This makes it fast for BFS and DFS since we can easily access all connected nodes. Compared to an adjacency matrix, which uses a full grid of  $V \times V$  spaces, an adjacency list saves a lot of memory, especially when the graph is not very connected. Edge lists are simpler because they only store edge pairs, but they are slower for finding neighbors, which makes them less suitable for traversal 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.

- The graph is undirected, meaning every edge goes both ways. When we add an edge between two nodes, the method automatically stores connections in both directions. This is why the `add_edge()` method adds `v` to `u` and also adds `u` to `v`. To make the graph directed, I would remove the second append so that edges only go in one direction. The BFS and DFS functions would still work, but the results could be different because some nodes might no longer be reachable in the opposite direction. Directed graphs are useful for modeling one-way systems like web links or road networks with one-way streets.

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.

- One real world problem that can be solved using graphs is social network analysis. Users can be nodes, and connections can be edges. BFS can help find the shortest connection path or suggest “friends of friends.” To support this fully, I may need to add weighted edges or make the graph directed for “follow” systems. Another example is navigation in maps, where locations are nodes and roads are edges. BFS can find shortest paths in simple cases, while DFS can explore possible routes. To make the code suitable for real maps, I would need to add weights to edges and include algorithms like Dijkstra’s or A for more accurate route finding.

## IV. Conclusion

In this activity, I learned how graphs work and how they can be represented in Python. I understood that a graph is made up of vertices and edges, and these connections help show relationships between different objects. By using a Python dictionary, it becomes easy to store and work with graph data. I also became more familiar with how undirected graphs connect nodes in both directions. Overall, this lesson helped me understand the basic concepts and functions of graphs, which are important for many computer science problems and real-world applications.