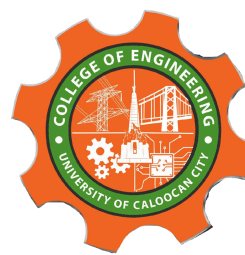




UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Balaoro, Judge Wayne 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.

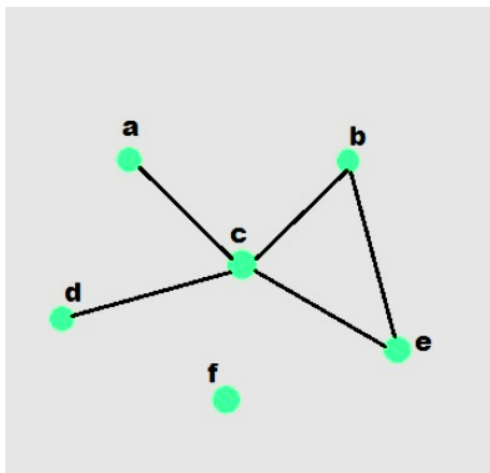


Figure 1. Sample graph with vertices and edges

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.

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.

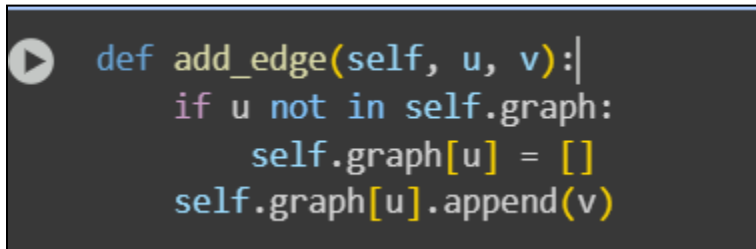
The BFS is like having everyone explore a place one step at a time, we check all paths of length one, and all length paths of two and so on, making us find the closest exit. It uses a queue to remember which paths to explore next. While the DFS is like just one single person exploring a place. going into a dead end before trying another route. This time it uses a stack to remember which turns to go back to. This is why it's often implemented with simple and recursive code.

3.

The graph implementation uses an adjacency list, which is a balanced way to store connections. It's like a contact list, where each person's entry is a vertex and it has a list of only the people they are friends with, which is the neighbor. This approach is memory-efficient since it's a graph where it has many vertices but not that many edges so it doesn't waste space on connections that don't exist. Comparing it to the alternative which is the adjacency matrix, it's like a spreadsheet with every vertex listed on both the rows and columns. While this matrix lets you check for specific almost instantly, it becomes very large and wasteful for graphs with a ton of edges. The last one is a much more simple method which is the edge list. It's a straightforward list of all the pairs of vertices that are connected like (A, B), (B, C). Although easy to set up it is generally the slowest for finding a vertex neighbor.

4.

The graph is currently implemented as undirected, this is directly enforced by the `add_edge` method. This method ensures that every connection is corresponding by creating a two way path like when edge from `u` to `v` is added and vice versa is also automatically created.

A code editor window with a dark background and light-colored text. It contains a Python function definition for adding a directed edge to a graph. The function is named 'add_edge' and takes 'self', 'u', and 'v' as arguments. It checks if 'u' is not in 'self.graph'. If true, it initializes 'self.graph[u]' as an empty list and then appends 'v' to it. A play button icon is visible on the left side of the code block.

```
def add_edge(self, u, v):  
    if u not in self.graph:  
        self.graph[u] = []  
    self.graph[u].append(v)
```

Figure 2. Directed modification

To support a directed graph, where the connections are one way, the method `add_edge` must be modified to create a single, directional link. This is attainable by removing the line that establishes the reverse connection. This modification will now follow explicit direction of edges which is critical for systems with asymmetric relationships such as follower-based social media.

5.

Two real world problems that graphs can solve is finding a friend suggestion on social apps and also calculating the routes in a GPS system. For social apps, the undirected graph is a great fit since users are vertices and friendship are edges. Running a BFS is the perfect way to find “friends of friends” to recommend a new connection. For GPS, the implementation needs a significant change since the graph must become directed to handle one-way streets and weighted to store data like you would need to implement a more advanced pathfinding algorithm to find the fastest or the shortest route.

IV. Conclusion

This laboratory activity provided me with a foundation of graph theory by implementing a graph data structure and its traversal algorithm such as BFS (Breadth-First Search) and DFS (Depth-First Search). Through this I gained a clear understanding on how BFS uses a queue for level by level search while the DFS uses a stack to explore as deeply as possible. I was also able to compare the adjacency list to its alternative adjacency matrix and also to a much simpler method which is the edge list. I also learned through hands-on modifying¹ the code which is the `add_edge` method to make it a directed graph.

References

- [1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.
- [2] GeeksforGeeks. (2025b, July 11). *Difference between BFS and DFS*. GeeksforGeeks.
<https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs>
- [3] *W3Schools.com*. (n.d.-c). https://www.w3schools.com/dsa/dsa_algo_graphs_traversal.php