UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Uy, Junichiro H.

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.

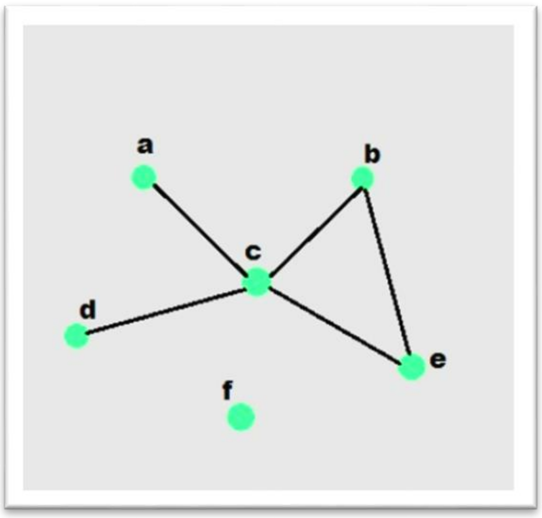

Figure 1. Sample graph with vertices and edges

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. 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]
```

2. BFS or Breadth-First Search is checking starting from the start node then to its neigbours. From 1 node away it will check every connection, then after that it deques to check the other nodes each layer. It explores wide before deep. This uses Queues, specifically deque from the collection modules. DFS or Depth-First Search, it just remains on one path. Meaning it goes deep as soon as it starts, it doesn't go back and visit a neighbor of the node it previously visited, it just goes deeper and starts looking for another neighbor, and once it reaches the dead end, it backstrack to check if it missed any nodes. This uses stack data type.

3.

Adjacency List:

This checks the neighbors of each nodes, like how you would ask each person who their friends are. Example is if you ask node 0 its neighbors, it would say 1 and 2, then you ask node 1, and it says 2. So in a sense, this is a list in a list. This uses dictionaries.

Pros: It is very space efficient and provides the fastest way to look for connections each node.
Cons: To see if some nodes are connected, it has to check the entire list of that certain node. Meaning it's slower compared to others.

Adjacency Matrices:

To make this simple, this is like a grid layout, like a spreadsheet. You put each node in a fixed format, then use symbols or indicators to indicate if they are connected or not. Like if node 0 and node 1 is indicated as number 1, meaning they are connected, and node 1 and node 4 are indicted as number 0 because they aren't connected.

Pros: This is very fast to use, because it just has to look at the box of node 0 and 1 to check if they are connected.
Cons: It takes up a lot of space, because it has to repeatedly alot spaces each node and their corresponding possible connections. Like if there are 5 nodes, it would need 5 boxes each node to place indicators.

Edge List:

By far, this is the simplest of them all, but the most inefficient one. Because it puts every connection of each nodes one by one in a long list, like: (node 0, node 1) then so on.

Pros: This is simple and easy to understand.
Cons: It's awfully slow, because it has to check the whole list and pick out each connection one by one.

4. The implication is added when the function add_edge is used. When an edge is added between u and v, this executes two operations:

```
        """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] = []
```

This creates a two-way connection, because if you connect u to v, you also make way for v to u. Making it impossible to create a one-way connection. Essentially, making it an undirected graph.

Then if we want to make this a Directed Graph, we need to remove that one append, and only keep one:

```
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] = []

        if v not in self.graph[u]: #So we can avoid duplicating nodes
            self.graph[u].append(v)
```

This is enough, we don't need to change anything in the BFS and DFS. This just remove the two-way connection we had before. Meaning, we can connect a node to another node without the other node being connected to the node we used. Node 0 is connected to Node 1 while node 0 won't be added to node 1's list.

5. Two Real World Problems:
Facebook Friends Connections: Let's say each account is a node. We can use the add_edge function to make use of the undirected graph to show that account a is friends with account b, and account b is friends with account a. Now let's say that account b is friends with account c, if we want to make a graph with these three accounts, we can use BFS to find the shortest path from a to c, and this also guarantees that we can check every connection they have because we search widely not deeply.

University course pre-requisites: Since some subjects in courses requires you to finish previous subjects to take it, we can use a one-way connection or directed graph to indicate it. Let's say subject A is connected to subject B, but subject B doesn't contain subject A anymore because you are done tackling it, meaning you can’t retake it anymore. But if you didn’t take subject A, there’s no way for you to go to subject B. For this specific problem, we will use DFS because it is the ideal way to go with directed graphs. From subject A to subject B, it won’t look for anymore connections other than subject B, it will lock in on that and backtrack when it hit the dead end. To make this possible, we will use the code we modified earlier to make it possible with one-way connections.

IV. Conclusion

In conclusion, using directed and undirected graphs, and knowing what to use (BFS and DFS) can provide for efficient data accessing for different purposes. This lab teaches me how to visualize and use different algorithms to find and make connections to different individual nodes.

References

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