UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Mamanao, Kurt Marwin, C..

Instructor:
Engr. Maria Rizette H. Sayo

OCT, 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.

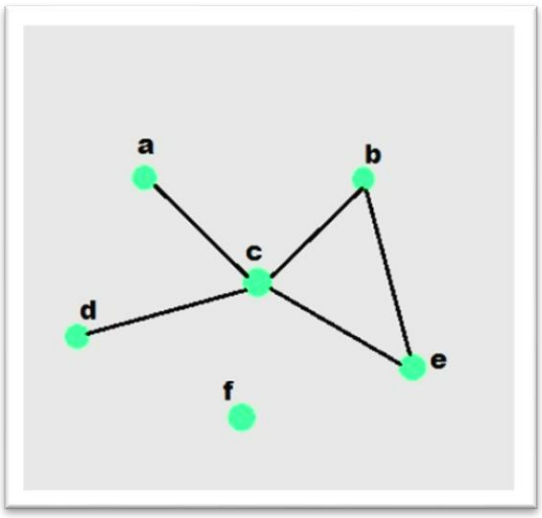


Figure 1. Sample graph with vertices and edges

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
```

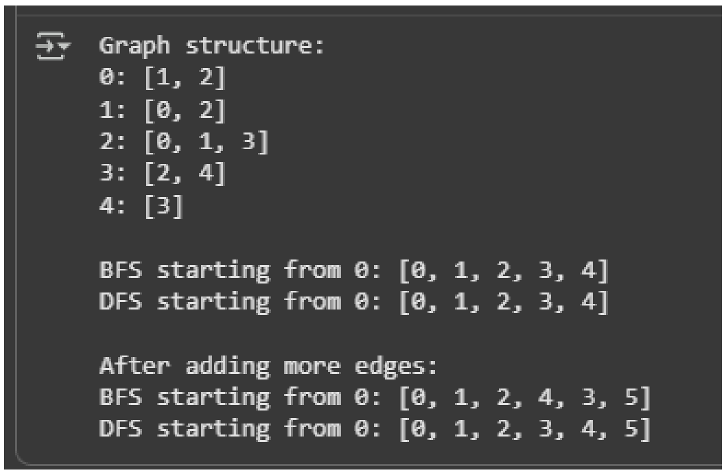
```
g = Graph()

# 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

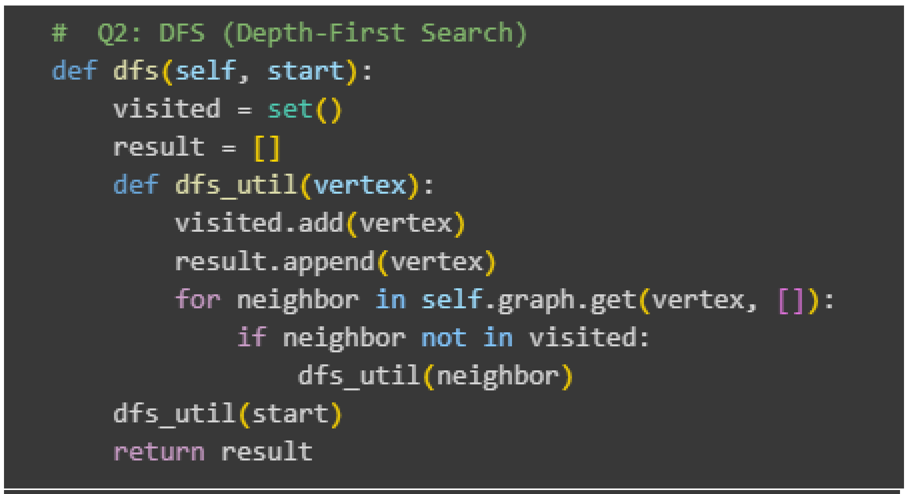
1. What will be the output of the following codes?



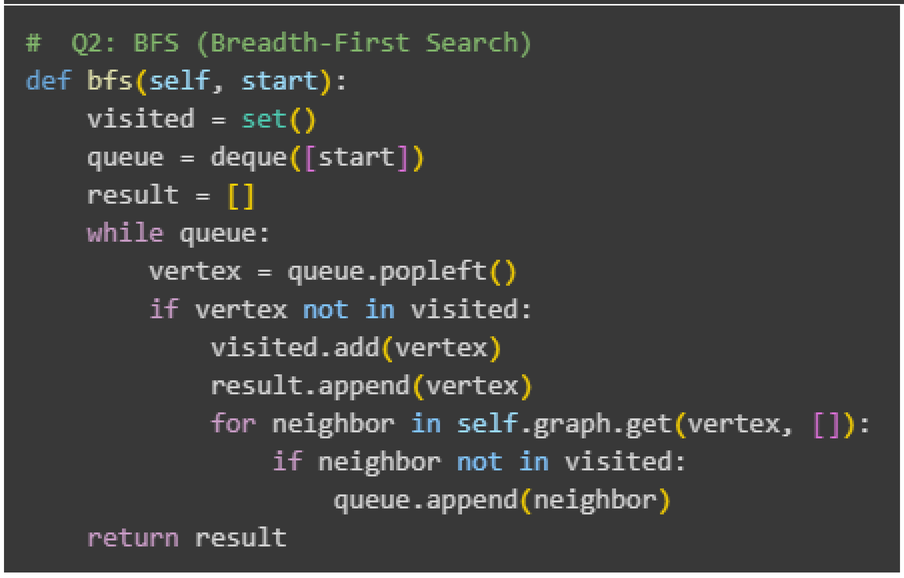
```
Graph structure:
0: [1, 2]
1: [0, 2]
2: [0, 1, 3]
3: [2, 4]
4: [3]

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]
```

-When I ran the code, it first showed all the connections between the nodes. After that, it displayed the BFS and DFS results starting from node 0. The BFS checks nodes step by step, while DFS goes deeper before moving to the next. When I added more edges, the search also included one more node (node 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?



```
#  Q2: DFS (Depth-First Search)
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
```



```
#  Q2: BFS (Breadth-First Search)
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
```

- For me, **BFS** starts from one node and visits all nearby nodes first before moving farther. It uses a **queue**, so it goes step by step. **DFS** goes deep into one path first before checking others. It uses **recursion** or a

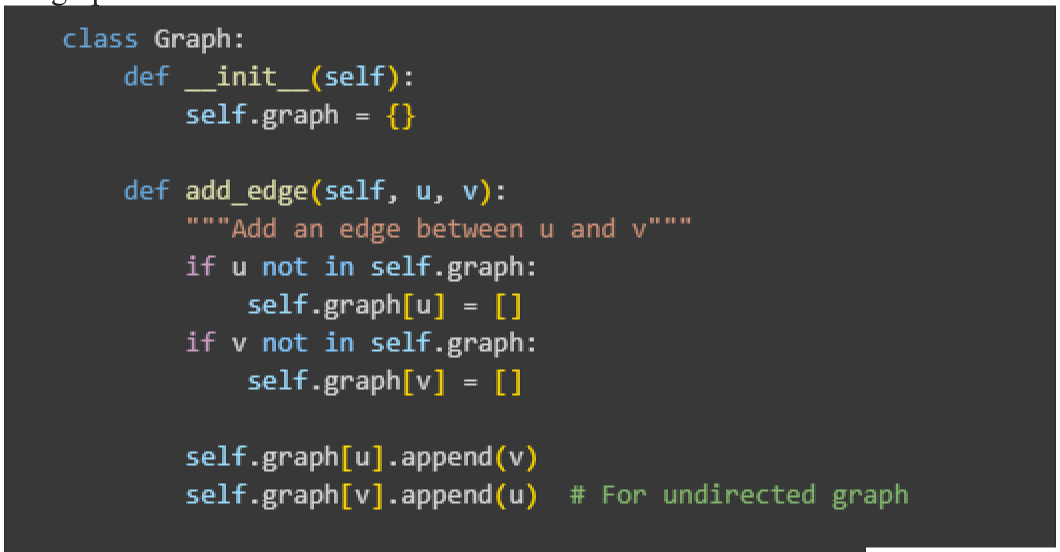
stack.
BFS is better for finding the shortest path, while DFS is good for exploring all possible routes, BFS feels like checking things layer by layer, while DFS feels like going down one way before coming back.

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.

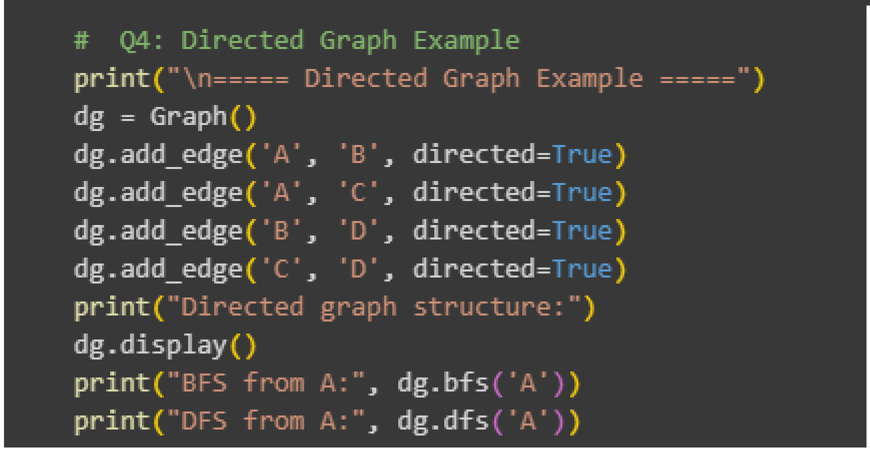
    The given code uses an **adjacency list**, where each node stores the nodes it connects to.
    An **adjacency matrix** uses a table of 1s and 0s to show if two nodes are connected, but it takes more space.
    An **edge list** just lists all the pairs of connected nodes.

    For me, the adjacency list is easier to understand and uses less memory. It only shows the connections that really exist, unlike a matrix that can waste space when there are few edges.

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.



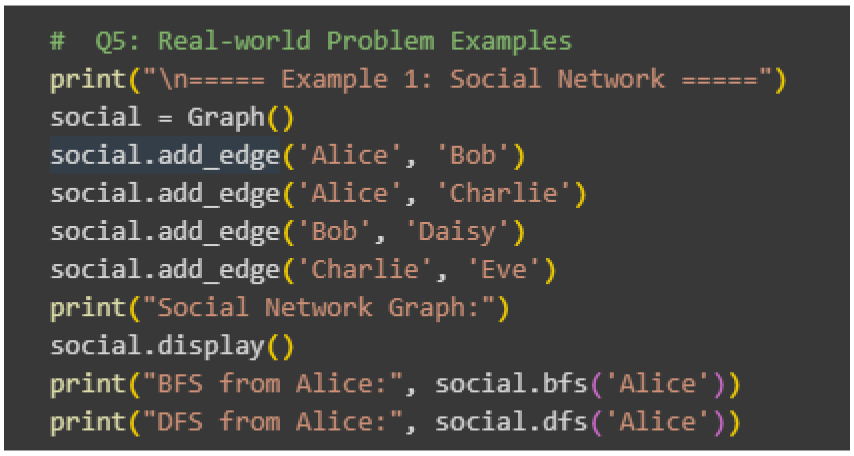
```python
class Graph:
    def __init__(self):
        self.graph = {}

    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
```



```python
#  Q4: Directed Graph Example
print("\n===== Directed Graph Example =====")
dg = Graph()
dg.add_edge('A', 'B', directed=True)
dg.add_edge('A', 'C', directed=True)
dg.add_edge('B', 'D', directed=True)
dg.add_edge('C', 'D', directed=True)
print("Directed graph structure:")
dg.display()
print("BFS from A:", dg.bfs('A'))
print("DFS from A:", dg.dfs('A'))
```

In the original code, when I add an edge between two nodes, it connects both ways. I changed the code so that when directed True, it only connects in one direction. This makes the graph work for both undirected and directed types, depending on what I need.

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.



```
#  Q5: Real-world Problem Examples
print("\n===== Example 1: Social Network =====")
social = Graph()
social.add_edge('Alice', 'Bob')
social.add_edge('Alice', 'Charlie')
social.add_edge('Bob', 'Daisy')
social.add_edge('Charlie', 'Eve')
print("Social Network Graph:")
social.display()
print("BFS from Alice:", social.bfs('Alice'))
print("DFS from Alice:", social.dfs('Alice'))
```

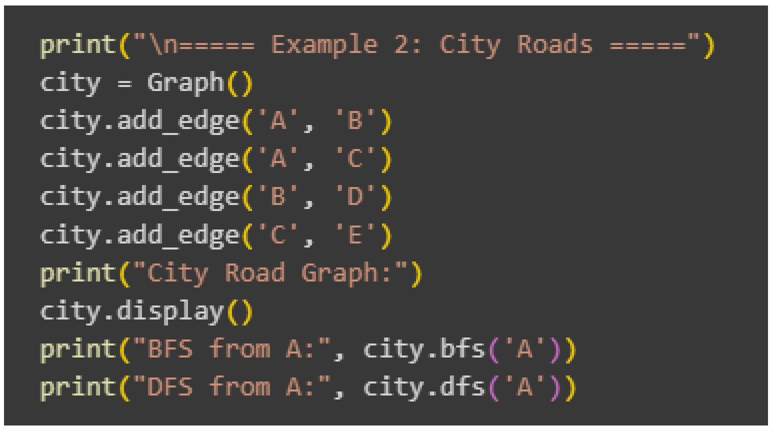
Social Network Example

Vertices: People (e.g., Alice, Bob, Charlie, etc.)

Edges: Friend connections

BFS Use: To find mutual friends or the shortest connection between two users.

DFS Use: To explore all people connected in one social group.



```
print("\n===== Example 2: City Roads =====")
city = Graph()
city.add_edge('A', 'B')
city.add_edge('A', 'C')
city.add_edge('B', 'D')
city.add_edge('C', 'E')
print("City Road Graph:")
city.display()
print("BFS from A:", city.bfs('A'))
print("DFS from A:", city.dfs('A'))
```

City Road Map Example

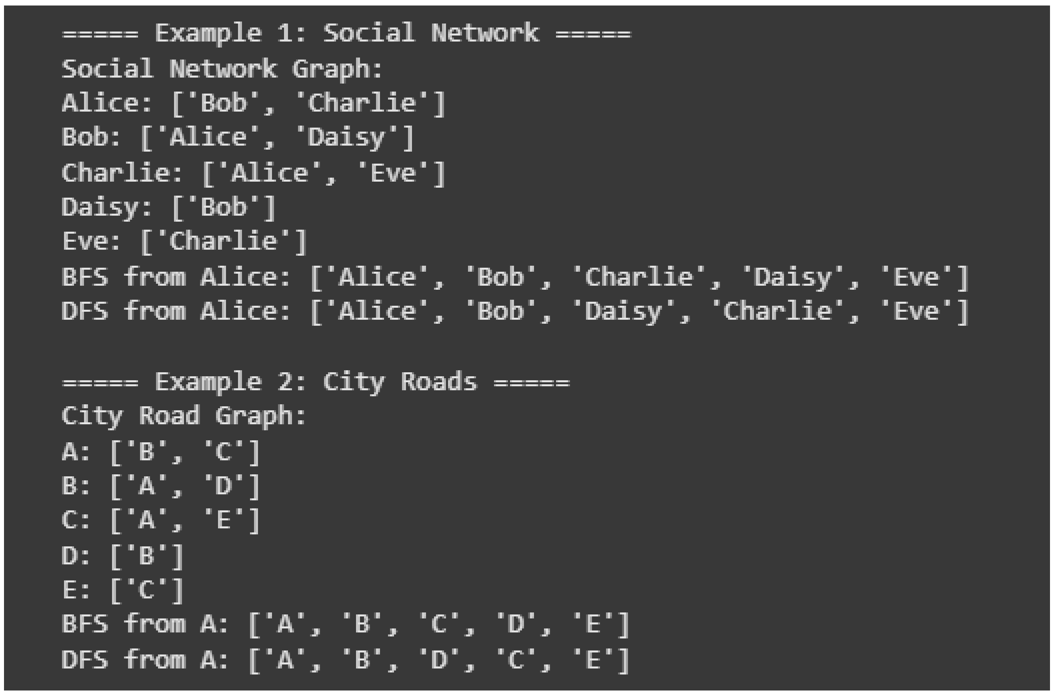
Vertices: Cities or intersections

Edges: Roads between them

BFS Use: To find shortest path or least number of roads to reach a destination.

DFS Use: To explore every possible route.

OUTPUT



```
===== Example 1: Social Network =====
Social Network Graph:
Alice: ['Bob', 'Charlie']
Bob: ['Alice', 'Daisy']
Charlie: ['Alice', 'Eve']
Daisy: ['Bob']
Eve: ['Charlie']
BFS from Alice: ['Alice', 'Bob', 'Charlie', 'Daisy', 'Eve']
DFS from Alice: ['Alice', 'Bob', 'Daisy', 'Charlie', 'Eve']

===== Example 2: City Roads =====
City Road Graph:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B']
E: ['C']
BFS from A: ['A', 'B', 'C', 'D', 'E']
DFS from A: ['A', 'B', 'D', 'C', 'E']
```

. For me, graphs are easier to understand because they can show how things are connected or related. BFS is useful when I need to find the shortest path, and DFS is good when I want to go through all possible paths in the graph

IV. Conclusion

I had a hard time understanding graphs at first, especially how BFS and DFS work. After running the code and seeing the results, I slowly understood how they move through the graph. I also learned how to change the code for directed graphs. Even though it was confusing at first, this activity helped me understand how graphs and their algorithms really work.

References

[1] GeeksforGeeks, “Graph and its representations,” [Online]. Available:
https://www.geeksforgeeks.org/graph-and-its-representations/