UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
AMPONG, J-KEVIN L.

Instructor:
Engr. Maria Rizette H. Sayo

Month, DD, YYYY

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.

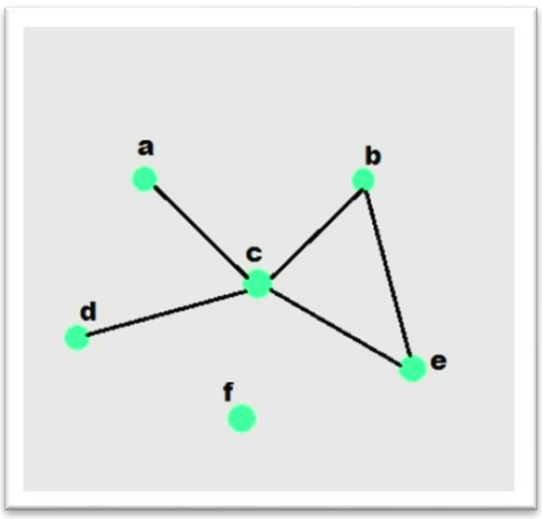


Figure 1. Sample graph with vertices and edges

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]
```

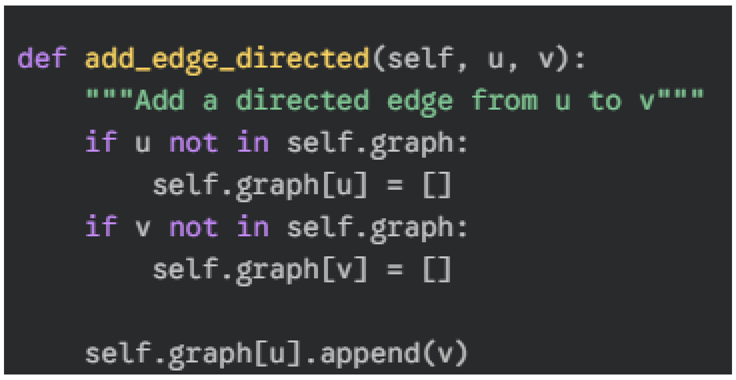
```
3: [2, 4]
4: [3]

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.
BFS and DFS are two ways to explore a graph, but they use different strategies. BFS (Breadth-First Search) is like searching outwards in circles, exploring all nearby nodes before moving to the next "level" further away. It uses a Queue (First-In, First-Out), which is why it always finds the shortest path in terms of the number of steps. Its implementation is iterative, meaning it uses loops and avoids the risk of running out of memory from too many nested calls. DFS (Depth-First Search) is like exploring a single path as far as it goes before backtracking. It uses Recursion and relies on the Call Stack (Last-In, First-Out), making the code often simpler and more elegant. However, a deep path in a large graph risks a stack overflow error. While their approach and primary data structures (Queue vs. Stack/Recursion) are different, both are efficient, taking about the same amount of time, O(V + E) to explore the entire graph.

3. The adjacency list is the chosen method for implementing the graph, offering an efficient way to store connections. Think of it as a contact list: each vertex (like a person) has a list of only its neighbors (the people they are directly connected to). This is memory-efficient for sparse graphs (graphs with many vertices but relatively few edges) because it avoids storing non-existent connections.

4. The current graph implementation is explicitly undirected because the add_edge(u, v) method automatically creates connections in both directions by adding **v** to **u**'s list and **u** to **v**'s list. This design choice simplifies the model, making it suitable for symmetric relationships like friendships or two-way roads, where if you can go from A to B, you can also go from B to A. To modify the code to support a directed graph, the key change is within the edge addition method. You would rename it (e.g., add_directed_edge) and remove the line that adds the reverse connection: **v** would be added as a neighbor of **u**, but **u** would not be automatically added as a neighbor of **v**. This means a traversal algorithm, like BFS or DFS, could move from **u** to **v**, but the path back from **v** to **u** would only exist if a separate, explicit edge from **v** to **u** were added.

This modification is essential for modeling asymmetric relationships, such as dependencies (Task A must precede Task B) or web links (a page links out, but isn't necessarily linked back to). The graph's behavior changes significantly, as reachability is no longer symmetric, making the order of nodes in the traversal results highly dependent on the direction of the edges.



```
def add_edge_directed(self, u, v):
    """Add a directed edge from u to v"""
    if u not in self.graph:
        self.graph[u] = []
    if v not in self.graph:
        self.graph[v] = []

    self.graph[u].append(v)
```

5. Two real-world problems that can be effectively modeled using the provided graph structure are Social Network Analysis and Finding Optimal Routes (Navigation).
For Social Network Analysis, the graph models users as vertices and friendships (or followers) as undirected edges. The provided implementation is immediately suitable since friendships are typically mutual. We would use the BFS algorithm to find the "degrees of separation" between any two users, as BFS guarantees the shortest path (fewest connections). To solve this fully, the existing bfs method would need an extension to store the predecessor of each visited node, allowing us to reconstruct and display the actual path of connections once the target user is found. For Finding Optimal Routes in a navigation system (like finding the best path between two cities), the cities or intersections are the vertices, and the roads are the edges. While the current code is undirected (assuming two-way roads), this is a good start. However, real-world routes usually have costs (distance, time, or traffic), making the graph weighted. A significant modification would be necessary to convert the adjacency list to store tuples of (neighbor, weight). Furthermore, instead of BFS or DFS, we would need to implement an algorithm like Dijkstra's Algorithm or A Search to find the shortest path based on these weights, not just the number of stops. While BFS and DFS could still be used for simple reachability checks, they wouldn't be sufficient for finding the *optimal* (cheapest) route.

IV. Conclusion

By using a graph data structure and associated traversal algorithms, such as BFS (Breadth-First Search) and DFS (Depth-First Search), this lab exercise gave me a basic understanding of graph theory. This helped me to clearly comprehend how the DFS employs a stack to explore as thoroughly as feasible, while the BFS uses a queue for level-by-level search. Additionally, I was able to compare the adjacency list to both its alternative adjacency matrix and the edge list, which is a much more straightforward approach. Additionally, I gained practical experience by altering the code using the add_edge technique to create a directed graph.

References

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