Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Asugas, Kenneth R.

Instructor:
Engr. Maria Rizette H. Sayo

10, 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.
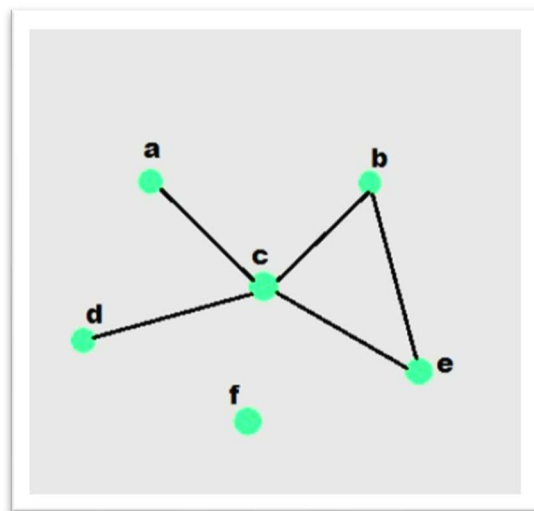

Figure 1. Sample graph with vertices and edges

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.

```python
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
```

2

```
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

Answers:

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 employs a queue and traverses nodes level by level, whereas DFS employs recursion to go deep and then backtracks. BFS is iterative and identifies the shortest path quickly, whereas DFS ventures out all the possible paths. Both have a time complexity of $O(V + E)$, although BFS consumes more memory and DFS has the possibility of stack overflow for big graphs.

3. An adjacency list stores only existing edges, making it space-efficient and faster for sparse graphs. An adjacency matrix uses more space but allows quick edge lookups. An edge list is the simplest form but slower when checking connected vertices.

4. Every edge in the undirected graph connects in both directions (u–v and v–u). The graph can be made directed for one-way links, such as websites or traffic systems, by removing the reverse connection in the add_edge method so that it only adds u → v.

5. One example is a social network where users are nodes and friendships are edges; BFS can find the shortest connection path. Another is a road map where intersections are nodes and roads are edges; BFS or Dijkstra's algorithm can be used to find the shortest route between two points.

IV. Conclusion

This lab report illustrated the use and traversal of graphs in Python. The experiment yielded real-world experience of how Breadth-First Search (BFS) and Depth-First Search (DFS) work on non-linear data structures.

By coding and analyzing, it was noted that BFS and DFS were very different in their order of traversal, data structure employed, and best applications. Adjacency list representation was efficient in storing and traversing graph data, particularly in sparse scenarios.

It is important to comprehend these basic graph algorithms in order to create sophisticated applications in networking, optimizing routes, social analysis, and artificial intelligence.

References

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