UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:
Enverzo, Kyle Andrey D.

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.

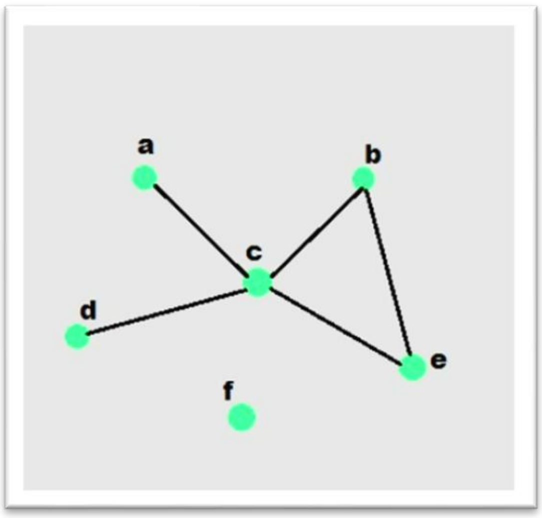


Figure 1. Sample graph with vertices and edges

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)}")
```

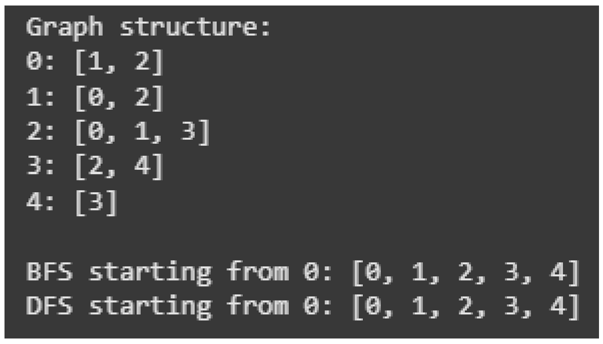
Questions:

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

ANSWER:

1.

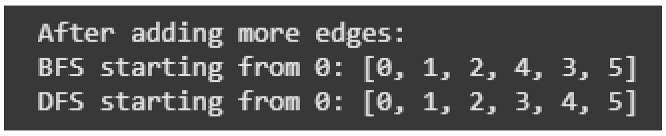


```
Graph structure:
0: [1, 2]
1: [0, 2]
2: [0, 1, 3]
3: [2, 4]
4: [3]

BFS starting from 0: [0, 1, 2, 3, 4]
DFS starting from 0: [0, 1, 2, 3, 4]
```

Figure 1 Output

The graph is such that nodes 0 to 4 are connected. BFS being a level order traversal of nodes starts from 0 and thus it first explores 1 and 2 and then 3 and 4. DFS also initiates at 0 but it goes deeper along a path before it backtracks hence it gives the same output as the graph is small and fully connected.



```
After adding more edges:
BFS starting from 0: [0, 1, 2, 4, 3, 5]
DFS starting from 0: [0, 1, 2, 3, 4, 5]
```

Figure 2 Output

New edges connect 4–5 and 1–4, expanding the graph. BFS now visits each level of vertices in order, reaching 5 last. DFS goes deep through one path before returning, which changes the visiting order compared to BFS.

2. Both the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms aim to get to all vertices of a graph, but they differ in their approaches. BFS is a queue-based method and hence after it has explored the neighbors of a node it goes further by one level. DFS, however, employs recursion (or a stack) to dive as far as possible along a path before it backtracks. BFS is a loop-based one and can be used to find the shortest path in unweighted graphs, whereas DFS is a recursive one and can be used to enumerate all possible paths or verify the graph's connectivity. BFS can sometimes hold a large amount of memory if the queue stores a large number of neighbors, whereas DFS can be a small memory-consuming way but in very large graphs it can go too deep.

3. The code graph employs adjacency list representation that keeps each vertex with the vertices to which it is directly connected. This way is good for graphs with less number of edges as it only keeps the existing connections. Whereas an adjacency matrix employs a 2D grid to represent all possible connections, which makes it faster to check if two vertices are connected but uses more memory. An edge list just records all edges as pairs of vertices, thus saving space but slowing down the search for connections. In general, the adjacency list is the least memory-consuming one in the majority of the real-world graph scenarios.

4. Simply put, 'the code describes an undirected graph' means that an edge connects two vertices in both ways. If there is an edge A to B, there must also exist an edge from B to A. Therefore in this design a traversal can wander across all these connected nodes freely. All that is required to convert this code into a directed graph is to remove that one line adding the reverse edge so that these connections would run only one way. Thus, this changes the nature of the graph to directional, making some of the nodes become unreachable and requiring BFS or DFS to only traverse those paths that follow the provided direction.

5. Graphs are capable of representing most real-world situations. One example here is a social network - where every people become a vertex and each friendship becomes an edge. BFS can find out the shortest path to connect two people; on the other hand, DFS can make exploration of all possible paths of friends. Another example could be pathfinding in maps or mazes. Each location is modeled as a vertex, while every road or path has an edge. BFS can find the shortest route and DFS would check all possible routes. Furthermore, to increase the realism of the models, we can put weights on the edges of the network (in terms of distance or cost) and apply advanced algorithms such as Dijkstra's or A* for more precise outputs.

IV. Conclusion

In this experiment, we implemented the graph actions using Python and analyzed the methods that were of particular interest to us-Breadth First search (BFS) and depth first search (DFS)-by writing a program that delivers a real-world representation of a graph from an adjacency list that stores connections between nodes very efficiently. Hence, the results confirm how BFS travels through nodes level by level, whereby you could get the least distance in a query; while DFS is going to start with deeper paths first and would basically proceed with it for connectivity and component analysis. This helped us understand undirected and directed graph structures as different ways to model and traverse relationships in data, either bi-directional or one-way links. This knowledge is extremely relevant in computer engineering because graph algorithms are used in major areas such as network routing, circuit design, data communication, and embedded system optimization.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA: Addison-Wesley, 2011.

[3] M. A. Weiss, *Data Structures and Algorithm Analysis in Python*, 2nd ed. Upper Saddle River, NJ: Pearson, 2014.

[4] D. E. Comer, *Computer Networks and Internets*, 6th ed. Boston, MA: Pearson, 2014.

[5] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Upper Saddle River, NJ: Pearson, 2011.