Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Ramos, Jan Lawrence M. | Engr. Maria Rizette H. Sayo |

October 18, 2025

# 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.

A diagram of a triangle with green dots

AI-generated content may be incorrect.

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

# Methods

* 1. Copy and run the Python source codes.
  2. If there is an algorithm error/s, debug the source codes.
  3. 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.

# Results

**Answers:**

**Question 1:**

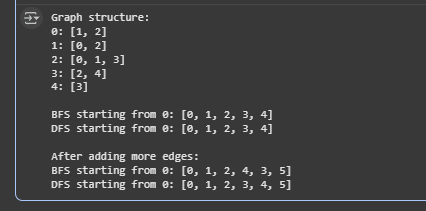


Figure 1: This figure shows the output of the debugged code.

**Question 2:**

**Data Structures:**

* BFS uses Queue (FIFO - First In First Out)
* DFS uses Stack (recursion = LIFO - Last In First Out)

**How they work:**

* BFS: Like spreading water - goes to all neighbors first
* DFS: Like going through maze - follows one path as far as possible

**Code difference:**

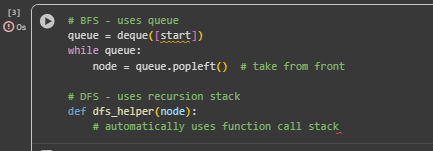


Figure 2: This figure shows the difference of BFS and DFS.

**Advantages:**

* BFS: Good for shortest path, level order
* DFS: Uses less memory, good for path finding

**Question 3:**

**Current: Adjacency List**

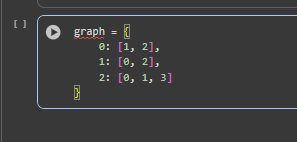


Figure 3: This shows the current: Adjacency list.

* Good for sparse graphs (few edges)
* Easy to add nodes
* Fast to get neighbors

**Alternative: Adjacency Matrix**

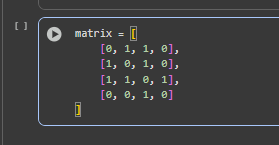


Figure 4: This shows the alternative way.

* Good for dense graphs (many edges)
* Fast to check if edge exists
* Uses more memory

**Question 4:**

**Current: Undirected Graph**

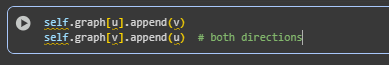


Figure 5: This shows the Undirected Graph.

* Edge works both ways

**Directed Graph Modification:**

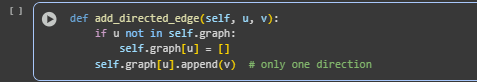


Figure 6: This shows the Directed Graph Modification.

* Edge goes one way only

**Question 5:**

**Example 1: Social Network**

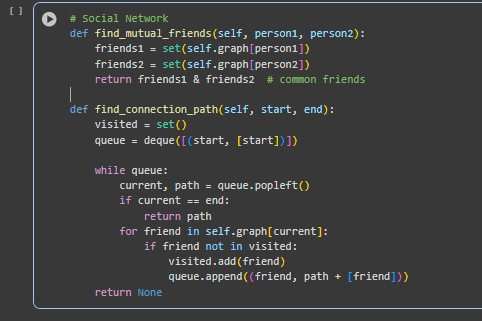
****

Figure 7: This shows the example of social network

**Example 2: Course Prerequisites**

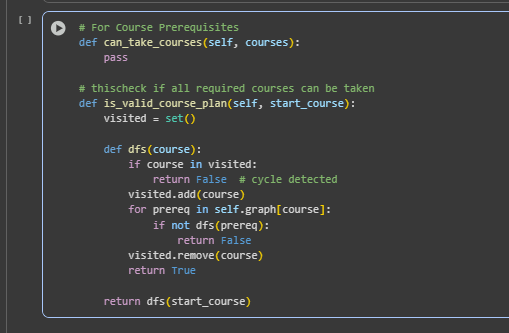
****

Figure 8: This shows the example of Course Prerequisites.

**Why BFS/DFS useful:**

* BFS: Find shortest path between people (social media)
* DFS: Check course prerequisites (academic planning)
* Both: Network analysis, recommendation systems

**Extra algorithms needed:**

* Shortest path (Dijkstra)
* Cycle detection
* Topological sort
* Connected components

# Conclusion

This laboratory work successfully implemented and analyzed the graph data structures and algorithms using Python. The graph class demonstrated practical implementation of adjacency list representation with support for both BFS and DFS traversal methods. Through this implementation, we gained hands on understanding of graph theory concepts and their real world applications. The comparison between BFS and DFS highlighted their distinct characteristics. BFS using queue based level order traversal suitable for shortest path finding, while DFS employing recursive depth first exploration better for path finding and connectivity analysis. The undirected graph design proved effective for modeling symmetric relationships, though modifications for directed graphs were also discussed. This work provided valuable insights into graph representations, traversal algorithms, and their computational complexities, establishing a solid foundation for more advanced graph algorithms and applications in computer engineering.

**References**

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

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. "Introduction to Algorithms," MIT Press, 3rd edition, 2009.