Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Nerio, Hannah Grace A. | 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

A white background with black text

AI-generated content may be incorrect.

Figure 1 Screenshot of program

Questions:

* + 1. What will be the output of the following codes?A white background with black text

       AI-generated content may be incorrect.
    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?

\*Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental traversal algorithms with distinct characteristics. BFS uses a queue data structure to explore nodes level by level, while DFS uses a recursive approach that relies on the system’s call stack to traverse deep into one branch before backtracking. Both algorithms share the same complexity of O(V + E), where V is the number of vertices and E is the number of edges, but they differ in space complexity and practical use. BFS tends to use more memory because of its queue but guarantees the shortest path in unweighted graphs. On the other hand, DFS uses less memory and is suitable for exploring paths, detecting cycles, and solving problems that require full exploration of possible routes. While BFS is iterative, DFS is recursive, which makes BFS better for shortest-path problems and DFS better for deep or exhaustive searches. However, the recursive nature of DFS may lead to stack overflow in very large graphs, which is an important limitation to consider.

* + 1. 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 graph implementation provided uses an adjacency list representation through a dictionary where each vertex maps to a list of its neighbors. This approach is efficient, especially for sparse graphs, because it only stores existing edges rather than maintaining a full matrix of possible connections. Compared to alternative representations, the adjacency list offers excellent space efficiency and faster traversal of connected nodes. An adjacency matrix, in contrast, uses a two-dimensional array where each cell indicates the presence or absence of an edge. While this allows constant-time edge lookups, it consumes O(V²) space, making it inefficient for large, sparse graphs. Another option, the edge list, simply stores pairs of connected vertices, which is simple to implement but slow to search when finding all neighbors of a vertex. Overall, the adjacency list strikes the best balance between space efficiency, clarity, and flexibility, making it ideal for dynamic graphs where edges are frequently added or removed.

* + 1. 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.

\*In the implementation provided, the graph is undirected, as indicated by the add\_edge method, which appends both u → v and v → u to the adjacency list. This design ensures that every connection between two vertices is bidirectional. The implication of this approach is that traversals such as BFS and DFS will always be able to move freely in both directions between connected vertices, modeling symmetric relationships such as friendships or mutual connections. To modify the code to support directed graphs, the add\_edge method can be adjusted so that the edge is added only from u to v, unless specified otherwise. For example, adding a directed parameter allows control over whether the edge should be one-way or two-way. This modification would significantly affect traversal behavior, as the algorithms would then follow edge directions, potentially resulting in fewer reachable nodes. Directed graphs are essential for modeling one-way systems such as road networks, dependency structures, or web links.

* + 1. 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.

\*Graphs have numerous real-world applications across various fields, serving as powerful tools for representing and analyzing relationships among entities. In social networks, each user can be modeled as a node, while friendships or interactions are represented as edges. Algorithms such as Breadth-First Search (BFS) can efficiently identify the shortest connection path between two users or suggest new connections by exploring mutual friends. In computer networks, graphs are used to represent routers and their communication links, where Depth-First Search (DFS) and Dijkstra’s algorithm help determine optimal routing paths, detect network failures, and maintain efficient data transmission. Beyond these, graphs are also applied in fields like transportation systems, where cities or intersections are treated as nodes and roads as edges, allowing planners to find shortest or least-cost routes. Overall, graph algorithms like BFS and DFS are fundamental not only in academic studies but also in real-world systems that rely on connectivity, optimization, and relational analysis.

# Conclusion

In conclusion, the laboratory activity on the Implementation of Graphs provided a deeper understanding of how graph data structures can efficiently represent and analyze relationships between entities. Through the Python program, the use of adjacency lists demonstrated a practical and memory-efficient way to store graph connections. By implementing both Breadth-First Search (BFS) and Depth-First Search (DFS), the experiment highlighted two fundamental traversal algorithms that differ in their exploration strategies yet share the goal of systematically visiting all vertices in a graph. The results showed how BFS explores nodes level by level, making it ideal for finding shortest paths, while DFS explores deeply along each branch, which is effective for pathfinding and connectivity analysis. Furthermore, the activity illustrated how real-world problems—such as social networks, computer networks, and transportation systems—can be represented and solved using graph algorithms like BFS, DFS, and Dijkstra’s. Overall, this experiment enhanced not only the theoretical understanding of graph structures but also the practical skills in implementing and applying them to real-world computational problems.

**References**

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

[2] P. Remis, M. J. Garzaran, R. Asenjo, and A. Navarro, “Exploiting social network graph characteristics for efficient BFS on heterogeneous chips,” University of Illinois at Urbana-Champaign, 2021.

[3] D. Grujic and B. Grujic, “Optimal routing in urban road networks: A graph-based approach using Dijkstra’s algorithm,” Applied Sciences, vol. 15, no. 8, p. 4162, 2025.