Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Caasi, Karl Benedict D. | 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

Answer  
1. A screenshot of a computer program

AI-generated content may be incorrect.

2. DFS (Recursive): Easier to code but can cause a stack overflow on large graphs.

BFS (Iterative)**:** Uses more memory for wide graphs but guarantees the shortest path.

Advantage   
 BFS: Good for shortest paths and spreading processes.

DFS: Good for exploring connectivity and deep searching.

A computer screen shot of a program

AI-generated content may be incorrect.  
3. It saves space, is easy to update, and works well for most real-world, sparse graphs.  
  
4. To make the graph directed, remove the reciprocal edge in add edge.  
This change affects reachability and makes the graph suitable for modeling one-way relationships such as web links, dependencies, or communication networks.

5. Graphs model real-world systems like social networks and maps.  
BFS finds the shortest paths, DFS explores routes deeply, and adding weights or directions makes the graph suitable for more complex problems.

**Conclusion**The Python application uses BFS and DFS to demonstrate how graphs function.  
While DFS uses recursion to delve deeper, BFS uses a queue to investigate nodes level by level. The graph is made simple and effective by the adjacency list, particularly for sparse or small networks. By modifying the add\_edge method, the graph can be made directed. By default, it is undirected, meaning connections flow both ways.

**Reference**   
 GeeksforGeeks. (n.d.). *Graph Data Structure and Algorithms*.  
Retrieved from: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

Programiz (n.d.). *Breadth-First Search (BFS) Algorithm*.  
Retrieved from: https://www.programiz.com/dsa/graph-bfs

Programiz (n.d.). *Depth-First Search (DFS) Algorithm*.  
Retrieved from: https://www.programiz.com/dsa/graph-dfs

W3Schools. (n.d.). *Python Data Structures – Graphs*.  
Retrieved from: https://www.w3schools.com/dsa/dsa\_data\_structures\_graphs.php