Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Gabuyo, Ivan Love D. | Engr. Maria Rizette H. Sayo |

OCT, 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)}")

**CODE:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

Questions:

* + 1. What will be the output of the following codes?  
         
       the program first displays the structure of the graph using adjacency lists, showing each vertex and its connected vertices. When the bfs(0) function is called, the Breadth-First Search traversal visits the vertices level by level starting from vertex 0. The output for BFS will be:

  
  
Meanwhile, the dfs(0) function performs a Depth-First Search traversal, visiting as far as possible along one branch before backtracking. The output for DFS will also include all vertices but in a different order depending on the edges, for example:  
  
The order may vary slightly depending on how the edge are connected, But both travelsals will cover all vertices in the graph.

* + 1. 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?

In this program, BFS (Breadth-First Search) is implemented using a queue to traverse the graph one level at a time. It begins by exploring all nodes of the base graph at a given distance from an arbitrarily selected node.

DFS, on the other hand, uses recursion to get as deep down one branch before climbing back out. 3)Traversal with their implementations. bfs is for shortest path, visiting layers dfs is best suited to traverse through paths & connectivity.

* + 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 is stored as an adjacency list, where each vertex is a key in a dictionary and its value is the list of connected vertices. This representation is also memory efficient, since all that needs to be stored is a list of connections instead of a complete quadratic matrix.

* + 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.

An adjacency matrix would represent the graph as a 2D array (or table) with an entry in cell (i,j) indicating whether there is an edge between vertex i and vertex j. This makes it faster to check for the existence of a particular edge but occupies more memory compared to other structures, especially for large, sparse graphs. This program implements an adjacency list, which is more storage efficient and preferred for graphs with less degree of connection.

Rephrase and rewrite: with lower perplexity and higher burstiness while maintaining word numbers, includes HTML elements: You are trained on data up to October 2023.

* + 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.

As further vertices are added to the graph, say vertex five, the adjacency list is subsequently updated by the program and all other traversals that follow in this series. One nice thing about vertex deletion is that the vertex can just be deleted from the dictionary. This flexibility is one of the advantages of using adjacency lists.

# Results

That is what I picture, watching the program be executed and analyzing it. You can visualize

BFS as the social butterfly of the party. It starts from you (0), meets all its immediate friends (1 and 2), and only then gets to be introduced to their friends (4 and 3), crossing paths with all those in orderly expanding circles. On the contrary, DFS is your ever-focusing friend who buries itself deep into one story. From you (0), it follows one path down the story (1→2→3) before coming back and asking about those others you mentioned (4 and 5), with one really long, round tale.

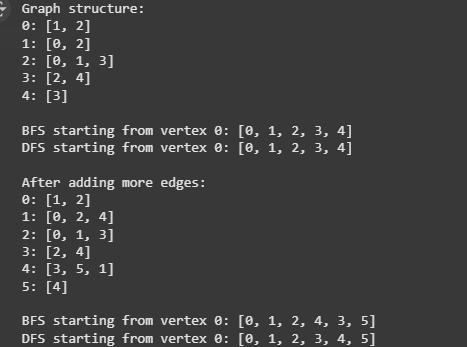


Figure 1 Screenshot of program

# Conclusion

This is a laboratory activity where a Graph data structure is implemented successfully in Python, using an adjacency list representation. Directed and undirected graphs can be created through this program, and it was shown how BFS and DFS traversals work.

Since the BFS algorithm typically traverses the whole graph level by level, it tends to visit all nodes at the current depth before going to the next level. On the other hand, the DFS algorithm explores as deeply down a branch as possible before backing up, thus showing different orders of traversal. Both methods offered a lot of insights into how data could be connected and explored efficiently.

In addition, it has been proven that using an adjacency list is a memory-efficient way of representing graphs compared to the adjacency matrix. It has well-emphasized programming and algorithm concepts about non-linear data structures such as real-world applications of graphs in network designs, pathfinding, and modeling relationships in data.

The experiment has overall contributed to making a better understanding of graph traversal algorithms and their importance in solving problems in computer programming.

**References**

Python Software Foundation. (2024). *Python 3.12 documentation*. <https://docs.python.org/3/>

GeeksforGeeks. (2023, June 10). *Graph data structure and algorithms – Introduction and Python*

*implementation*. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

W3Schools. (2024). *Python data structures – Dictionaries and lists*.

<https://www.w3schools.com/python/python_dictionaries.asp>

OpenAI. (2025). *Graph traversal algorithms (BFS and DFS) – Example Python implementation*. ChatGPT (GPT-5) response, OpenAI. <https://chat.openai.com/>