Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Calica, Ljay L. | Engr. Maria Rizette H. Sayo |

Oct, 10, 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:**

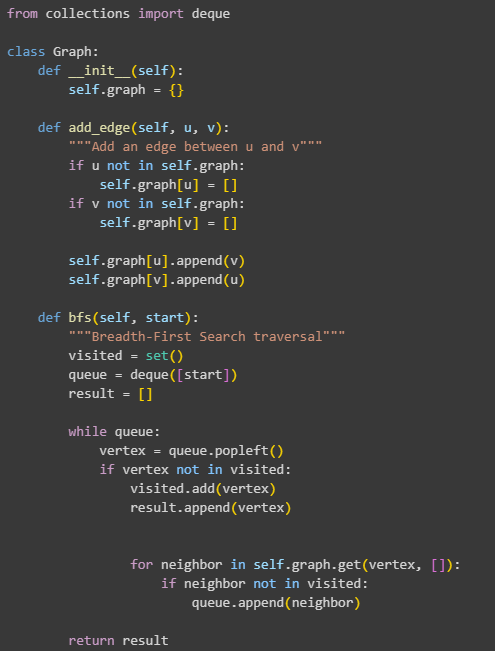


Figure 1 Screenshot of program



Figure 2 Screenshot of program

Questions:

* + 1. What will be the output of the following codes?

//The BFS result shows a level-by-level crawl from the start, visiting 0, then its direct neighbors 1 and 2, then their connections, resulting in [0, 1, 2, 4, 3, 5]. The DFS result shows a single path followed stubbornly to the end before backtracking, creating the long chain [0, 1, 2, 3, 4, 5].

* + 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?

//BFS uses a queue to methodically explore a graph level by level, guaranteeing the shortest path is found first, while DFS uses a stack (via recursion) to dive down one branch completely before backtracking, making it better for tasks like checking if a path exists. This fundamental difference makes BFS ideal for finding minimum steps in a network, whereas DFS excels at exploring all possible routes or navigating mazes.

* + 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 provided adjacency list representation, which uses a dictionary of lists, is highly efficient for sparse graphs (those with few edges) because it only stores existing connections, saving significant memory. In contrast, an adjacency matrix uses a V×V grid where each cell indicates a connection, making it much faster to check if a specific edge exists but consuming far more memory, which makes it practical only for dense graphs or when fast edge lookup is the primary requirement.

* + 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.

//The current design as an undirected graph means every connection is two-way, which is simple but unrealistic for one-way relationships like web links or follower systems. To support directed graphs, you would modify the add\_edge method to only create a connection from u to v without automatically adding the reverse, and then carefully update BFS and DFS to respect this one-way direction, fundamentally changing the traversal to reflect real-world asymmetrical relationships.

* + 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.

//Social networks and transportation maps are examples of real-world settings that this graph system may readily model. While DFS searches through whole social circles for friend referrals, BFS uses common friends to determine the shortest link path between two users in a friendship network. In a subway system, DFS finds every potential path from a starting point, while BFS determines the route with the fewest stops between stations. We would only need to give the points actual names and give the links optional weights, such as friendship strength or trip times, to adapt the code and make the abstract structure useful for daily tasks.

# Results

This is what I can visualized after analyzing and run the code. Imagine BFS as a social butterfly at a party. It starts with you (0), meets everyone you know directly (1 and 2), and only then gets introduced to their friends (4 and 3), ensuring it meets everyone in orderly, expanding circles.

DFS is your focused friend who latches onto one story. From you (0), they dive deep into one conversation thread (1→2→3), following it to the end before finally circling back to ask about the other people you mentioned (4 and 5), resulting in a long, continuous tale.

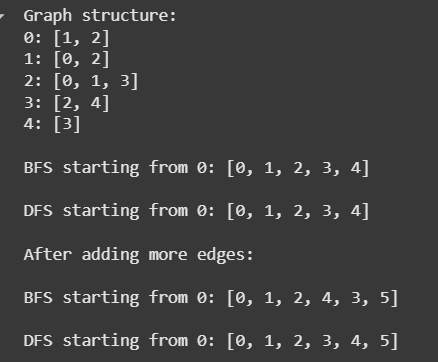


Figure 1 Screenshot of program

# Conclusion

By using Python to create graph data structures, this lab effectively illustrated their basic ideas. We learned how Breadth-First Search (BFS) and Depth-First Search (DFS) investigate a graph in essentially different ways by creating adjacency lists and writing traversal algorithms. Our comprehension of these fundamental non-linear data structures and the algorithms that go along with them was strengthened by the exercise, giving us a solid basis on which to solve increasingly challenging, real-world issues like pathfinding and network analysis.

**References**

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). The MIT Press.

[2] GeeksforGeeks. (2023, November 17). *Graph and its representations*. GeeksforGeeks.

<https://www.geeksforgeeks.org/graph-and-its-representations/>