Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Polestico,Paul Justine 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

Question #1 Answer:

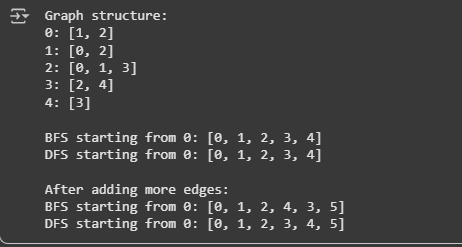


Figure 1 Screenshot of program

Question #2 Answer:

Based on the given graph code, Breadth-First Search (BFS) uses a queue, which follows the First-In, First-Out (FIFO) principle. This means nodes are visited in the order they were discovered, exploring the graph level by level.

In contrast, Depth-First Search (DFS) uses recursion, which behaves like a stack, following the Last-In, First-Out (LIFO) principle. This causes the traversal to go as deep as possible along one path before backtracking, visiting nodes sequentially in depth.

In terms of Time Complexity, they both have the O(V + E) complexity since each each node and edge is visited once.

The recursive nature of DFS allows repeated traversal while requiring less time to code, but can cause memory overflow. On the other hand, The iterative nature of BFS does not allow repeated traversal while requiring more time to code, but the chances of memory overflow are lessened and allow more breathing room for organized data.

Question #3 Answer:

In an adjacency list, the structure works similarly to a real-life dictionary where each category (node) contains a list of related books (neighbors). This makes it easy to see which books belong to which category—just like how you can quickly find all connections from a node.

However, if you want to check whether **two specific books are directly related**, you’d have to dig into each category’s list manually. Unlike a matrix, it’s not instant—you need to look up each node’s neighbors to confirm the connection.

In an adjacency matrix, imagine a seating chart where each student has a designated chair, arranged alphabetically. You can instantly check if two students are seated beside each other—just look at the intersection of their names on the chart.

However, if you have multiple seating charts (like different classrooms), it becomes harder to track relationships across rooms. The matrix is great for quick lookups within one space, but less flexible when comparing across multiple contexts.

Lastly, an edge list is like a guestbook that records who interacted with whom: Just a simple log of two classmates crossing paths. It captures the connection, but not the context.

So while we know they “talked,” we don’t know if they’re best friends, rivals, or secretly lovers pretending to be friends. In other words, the edge list tells us that a relationship exists, but not how deep or structured it is.

Question #4 Answer:

Since relationships between nodes are not explicitly defined in an undirected graph, I would modify the code to relate one node to another in a single direction by adjusting the method. This clearly states the nature of the connection.

For example, given nodes B and C, we can define their relationship by connecting B to C, indicating a directed edge from B to C. This way, the graph reflects not just the existence of a connection, but also its direction and intent—essential for modeling systems where flow or hierarchy matters.

Question #5 answer:

1.Classroom Chart Allocation

I would use an adjacency matrix to represent which students are enrolled in which programs, allowing for quick lookup and segregation. To better model the structure, I’d modify the code to support a bipartite graph, where one set of nodes represents students and the other represents programs. This is especially useful in cases like the Engineering program, where specific disciplines are tightly grouped.

• BFS: Finds the shortest path from a student to an available classroom—minimizing reassignment steps.

• DFS: Explores deeper connections to find a suitable classroom based on the student’s enrolled program.

• Additional Algorithms:

• Matching algorithms (e.g., for optimal student-program pairing)

• Constraint satisfaction (e.g., scheduling, capacity limits)

2. Order Routing in Jollibee

I’d use an edge list to document which customer interacted with which cashier—helpful for tracing complaints or order issues. For tracking the flow of orders throughout the day, I’d switch to a directed graph, where each node represents a station (e.g., counter, kitchen, fryer), and edges represent the sequence of operations.

• BFS: Identifies the shortest time path for an order to be completed.

• DFS: Useful for handling combo meals that require multiple stations and branching workflows.

• Additional Algorithms:

• Dijkstra’s Algorithm (for shortest processing time)

• Topological Sorting (to ensure tasks follow the correct sequence)

Conclusion

After analyzing the differences between DFS and BFS and their applications, I have come to realize the importance of knowing which one to use in different situations.

BFS is like scanning a room row by row,perfect for finding the shortest path or exploring things level by level. DFS, on the other hand, dives deep into one path before backtracking,ideal for exploring complex structures or solving puzzles where depth matters.

Choosing the right traversal isn’t just about speed,it’s about understanding the nature of the problem. Whether it’s routing orders in a fast-food chain or assigning students to classrooms, the way we traverse a graph can shape the outcome.

In short, BFS and DFS aren’t just algorithms,they’re strategies. And knowing when to use each one is like knowing whether to sprint or climb.

**References**

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