Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Hermosura, Leigh B. | 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

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

* The output for the code will be shown below:

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1 Output of the Program

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 graph traversal, Breadth-First Search (BFS) is an algorithm that starts with a node, then traverses all nodes adjacent to it. Once done visiting, it then traverses to the neighbors of the neighboring nodes, and so on. As per a queue, BFS follows a First-In First-Out (FIFO) order of traversal [1].
* Depth-First Search (DFS) on the other hand, starts with a node but explores as far along each branch as possible before backtracking. It processes the most recently discovered node first, continuing deeper into the graph before returning. DFS follows a Last-In First-Out (LIFO) order of traversal, similar to how a stack operates [2].

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.

* Adjacency list excels in sparse graphs, graphs that has relatively few edges. It can also traverse neighboring nodes quicker than the two alternatives [3].
* Adjacency matrices are efficient in dense graphs and is easy to implement due to its simplicity. It also has a faster access to graph edges but generally slower in traversal [4].
* Lastly, an edge list is also simple to implement and works well for sparse graphs. However, it loses its edge in dense graphs due to its longer traversal time when checking for connections [5].

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.

* For the code to support directed graphs, we need to modify the add\_edge function where instead of creating a two-way connection between u and v, we will just declare a one-way direction starting from u to v. Both BFS and DFS had no problem shifting to a directed graph. The graph behavior changes. Using undirected edges, we can form a connection between all nodes but directed edges may not reach all nodes if edges are one-way.

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.

* One real-world application is analyzing Wi-Fi connection latency between devices. Here, the vertices represent devices such as phones, computers, and other networked equipment, while the edges represent the connections between them, which can be either directed or undirected. Using BFS, we can identify all devices connected within the network and map the coverage area of a given router. Weighted edges can further represent signal strength between devices.
* Another example involves transportation routes and pathfinding. Vertices correspond to specific locations or landmarks used as reference points, and edges represent the routes between them. These edges can be directed or undirected depending on whether the routes are one-way or two-way. DFS can be applied to explore multiple possible routes and help find the shortest or most efficient path.

# Conclusion

Both BFS and DFS are fundamental graph traversal algorithms with distinct strategies suited to different applications. The choice of graph representation, adjacency list, adjacency matrix, or edge list, depends largely on the graph’s density and the specific operations required. Adjusting for directed or undirected edges further impacts graph behavior and traversal outcomes. These concepts find practical use in real-world problems such as analyzing network connectivity and optimizing transportation routes, demonstrating the versatility and importance of graph theory in computing and everyday applications.

**References**

[1] GeeksforGeeks, “Breadth First Search or BFS for a Graph,” *GeeksforGeeks*, Mar. 20, 2012. https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/

‌[2] GeeksforGeeks, “Depth First Search or DFS for a Graph,” *GeeksforGeeks*, Mar. 15, 2012. https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/

‌[3] GeeksforGeeks, “Adjacency List Representation,” *GeeksforGeeks*, Apr. 05, 2023. https://www.geeksforgeeks.org/dsa/adjacency-list-meaning-definition-in-dsa/

[4] GeeksforGeeks, “Adjacency Matrix Representation,” *GeeksforGeeks*, Apr. 24, 2024. https://www.geeksforgeeks.org/dsa/adjacency-matrix/

‌[5] “Edge list - (Data Structures) - Vocab, Definition, Explanations | Fiveable,” *Fiveable.me*, 2025. https://fiveable.me/key-terms/data-structures/edge-list (accessed Oct. 18, 2025).

‌