Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Monoy, Justin Rhey A. | 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

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1. Screenshot of program

* + 1. Graph exploration can be done in two different ways. First in Depth-First Search or DFS and Breadth-First Search or BFS. To determine the shortest path in unweighted graphs, BFS checks nodes level by level through a queue data structure. While this iterative method makes memory usage more predictable, it can use a lot of memory when handling shallow, wide graphs. On the other hand, DFS is memory-efficient for narrow, deep structures but could be challenging for very deep graphs because it uses recursion to go as far as possible along one path before looping back. While the time complexity of both methods is O (V + E), the structure of the graph can have a significant effect on their space complexity and practical performance.
    2. Every vertex in the present graph implementation points to a list of its neighbors through an adjacency list containing a dictionary. For most graphs, this method works well, particularly for graphs with few edges between vertices. Vertices with several connections may find it slow to look through a neighbor list to determine whether two particular vertices are related. While they consume more memory, alternatives like edge lists and adjacency matrices encode the graph as a 2D grid. Since the BFS and DFS algorithms must quickly access every neighbor of every vertex they visit, the adjacency list is the best option.
    3. Because the `add\_edge` function creates connections in both directions, creating relationships two-way, the current graph is undirected. As a result, BFS and DFS work naturally, and the graph structure is reduced. Change the `add\_edge` function to only add connections in one direction in order to convert to a directed graph. One-way connections, such as task dependencies or links to websites, are found in directed graphs. This affects the behavior of traversal algorithms, possibly making some vertices inaccessible from specific starting points, which has an effect on connection analysis and finding paths.
    4. Real-world problems including determining the fastest route between cities and suggesting friends on the internet could be represented with BFS. Through checking at their friends' connections first, BFS may be used to find similar friends within two to three degrees of separation on the internet. BFS may be used to find the shortest route between two cities for city route planning, however considerable adjustments are needed. These changes include managing one-way highways by turning the graph into a directed graph, applying Dijkstra's algorithm for shortest weighted routes, and giving edges weights for travel durations or road distances. In order to adjust for real distances or travel durations, other algorithms, such as Dijkstra's or A\* search, would be required.

# Conclusion

The BFS and DFS algorithms, which use vertices and edges to show connections between objects, are effective tools for analyzing and resolving challenging real-world problems. Using adjacency lists, which store the connections between vertices as lists of neighboring vertices and vertices as keys, Python dictionaries can efficiently represent graph structures. While DFS is better for path existence checks and deep exploration scenarios, BFS is particularly good at identifying the shortest paths and exploring level by level, which makes it ideal for friend suggestions and route planning. To choose the most efficient way, it is necessary to understand the consequences between different graph representations. The basis for solving a variety of problems related to computer networks, transportation systems, and social networks is given by these basic ideas.

**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] K. Moore, K. Jennison, and J. Khim, “Depth-First Search (DFS) | Brilliant Math & Science Wiki,” *brilliant.org*. <https://brilliant.org/wiki/depth-first-search-dfs/>

[3] “Graph Data Structure,” *www.programiz.com*. <https://www.programiz.com/dsa/graph>

[4] GeeksforGeeks, “Dijkstra’s Algorithm to find Shortest Paths from a Source to all,” *GeeksforGeeks*, Nov. 25, 2012. <https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/>

[5] GeeksforGeeks, “A\* Search Algorithm,” *GeeksforGeeks*, Jun. 16, 2016. https://www.geeksforgeeks.org/dsa/a-search-algorithm/

‌

‌