Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Balana, Jerkielle Roen O. | Engr. Maria Rizette H. Sayo |

October 25, 2025

# Objectives

Introduction

Depth-First Search (DFS)

* Explores as far as possible along each branch before backtracking
* Uses stack data structure (either explicitly or via recursion)
* Time Complexity: O(V + E)
* Space Complexity: O(V)

Breadth-First Search (BFS)

* Explores all neighbors at current depth before moving deeper
* Uses queue data structure
* Time Complexity: O(V + E)
* Space Complexity: O(V)

This laboratory activity aims to implement the principles and techniques in:

* Understand and implement Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms
* Compare the traversal order and behavior of both algorithms
* Analyze time and space complexity differences

# Methods

* + Copy and run the Python source codes.
  + If there is an algorithm error/s, debug the source codes.
  + Save these source codes to your GitHub.
  + Show the output
    1. Graph Implementation

from collections import deque

import time

class Graph:

def \_\_init\_\_(self):

self.adj\_list = {}

def add\_vertex(self, vertex):

if vertex not in self.adj\_list:

self.adj\_list[vertex] = []

def add\_edge(self, vertex1, vertex2, directed=False):

self.add\_vertex(vertex1)

self.add\_vertex(vertex2)

self.adj\_list[vertex1].append(vertex2)

if not directed:

self.adj\_list[vertex2].append(vertex1)

def display(self):

for vertex, neighbors in self.adj\_list.items():

print(f"{vertex}: {neighbors}")

2. DFS Implementation

def dfs\_recursive(graph, start, visited=None, path=None):

if visited is None:

visited = set()

if path is None:

path = []

visited.add(start)

path.append(start)

print(f"Visiting: {start}")

for neighbor in graph.adj\_list[start]:

if neighbor not in visited:

dfs\_recursive(graph, neighbor, visited, path)

return path

def dfs\_iterative(graph, start):

visited = set()

stack = [start]

path = []

print("DFS Iterative Traversal:")

while stack:

vertex = stack.pop()

if vertex not in visited:

visited.add(vertex)

path.append(vertex)

print(f"Visiting: {vertex}")

# Add neighbors in reverse order for same behavior as recursive

for neighbor in reversed(graph.adj\_list[vertex]):

if neighbor not in visited:

stack.append(neighbor)

return path

1. BFS Implementation

def bfs(graph, start):

visited = set()

queue = deque([start])

path = []

print("BFS Traversal:")

while queue:

vertex = queue.popleft()

if vertex not in visited:

visited.add(vertex)

path.append(vertex)

print(f"Visiting: {vertex}")

for neighbor in graph.adj\_list[vertex]:

if neighbor not in visited:

queue.append(neighbor)

return path

Questions:

1. When would you prefer DFS over BFS and vice versa?

If I had to choose one, I would prefer BFS, for it is more reliable in returning the shortest path and does not get lost in deep or sometimes infinite paths, which might happen in DFS.

1. What is the space complexity difference between DFS and BFS?

DFS has a space complexity of O(V), where V is the number of vertices. It primarily maintains the recursion stack. BFS has O(V) space complexity, too, but in general, it consumes more memory because it needs to store all vertices of the current level in the queue.

1. How does the traversal order differ between DFS and BFS?

DFS explores along one branch as far as it can before backtracking to other options. It follows a deep path-first pattern. This approach works like going down one route completely before switching to another. BFS checks all neighbors of a node before heading deeper. It follows a level-order pattern. This means visiting every node at depth one first, then all at depth two, and continuing that way.

1. When does DFS recursive fail compared to DFS iterative?

DFS-Recursive fails if the recursion depth is beyond the system's call stack limit, which often occurs for very large or deeply nested graphs. DFS-Iterative never suffers from this problem, since it uses a data structure as an explicit stack instead of relying on the system call stack; hence, it is more robust for large-scale graphs.

# Results

* + 1. If I had to choose one, I would prefer BFS, for it is more reliable in returning the shortest path and does not get lost in deep or sometimes infinite paths, which might happen in DFS.
    2. DFS has a space complexity of O(V), where V is the number of vertices. It primarily maintains the recursion stack. BFS has O(V) space complexity, too, but in general, it consumes more memory because it needs to store all vertices of the current level in the queue.
    3. DFS explores along one branch as far as it can before backtracking to other options. It follows a deep path-first pattern. This approach works like going down one route completely before switching to another. BFS checks all neighbors of a node before heading deeper. It follows a level-order pattern. This means visiting every node at depth one first, then all at depth two, and continuing that way.
    4. DFS-Recursive fails if the recursion depth is beyond the system's call stack limit, which often occurs for very large or deeply nested graphs. DFS-Iterative never suffers from this problem, since it uses a data structure as an explicit stack instead of relying on the system call stack; hence, it is more robust for large-scale graphs.

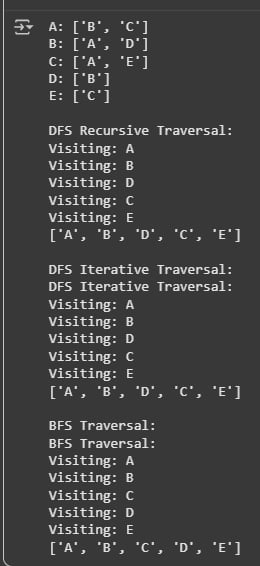


Figure 1 Screenshot of program

# Conclusion

In conclusion, the laboratory activity successfully implemented and showed the differences between Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms. Both methods can traverse all vertices in a graph, but their behavior and applications vary. DFS is best used for exhaustive searches and scenarios requiring deep exploration, while BFS is ideal in finding the shortest paths and exploring nodes level by level.

**References**

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