Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Hermosura, Leigh B. | 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?
2. What is the space complexity difference between DFS and BFS?
3. How does the traversal order differ between DFS and BFS?
4. When does DFS recursive fail compared to DFS iterative?

# Results

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1 Output of Graph, DFS, and BFS implementation

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

* DFS is preferred in solving puzzles or mazes as its backtracking process is suited in finding the best solution. BFS is efficient when finding the shortest path in unweighted graphs [1].

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

* Both DFS and BFS uses the same space complexity, O(V). In DFS, the space complexity depends on the maximum depth of recursion, while BFS depends on the maximum number of vertices in the queue for its space complexity [2].

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

* The traversal order differ between DFS and BFS as DFS explores connections as far as possible in each branch, while BFS visits all neighbors in the same depth first before proceeding deeper into the graph.

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

* Although DFS is easier to write it fails on deep graphs and can cause stack overflow. BFS on the other hand, has no risk of stack overflow and more memory-safe for large or deep graphs [3].

# Conclusion

DFS and BFS are fundamental graph traversal techniques, each with its own strengths and trade-offs. DFS is useful for exploring deep structures, solving puzzles, and backtracking, but its recursive implementation can fail on very deep graphs due to stack overflow. BFS, on the other hand, guarantees the shortest path in unweighted graphs and is safer for large or deep graphs, though it may require more memory to store all nodes at a given depth. Choosing between the two depends on the situation whether depth exploration or shortest-path efficiency is the priority.

**References**

[1] “When to Use BFS vs DFS in Graph Problems: A Comprehensive Guide – AlgoCademy Blog,” *Algocademy.com*, 2025. https://algocademy.com/blog/when-to-use-bfs-vs-dfs-in-graph-problems-a-comprehensive-guide/ (accessed Oct. 25, 2025).

[2] T. Sauce, “Time Complexity and Space Complexity of DFS and BFS Algorithms,” *Medium*, Oct. 21, 2023. https://techsauce.medium.com/time-complexity-and-space-complexity-of-dfs-and-bfs-algorithms-671217e43d58

[3] Arlan N, “DFS Iterative vs Recursive: When to Use Each and Why It Matters - Lodely,” *Lodely.com*, 2025. https://www.lodely.com/blog/dfs-iterative-vs-recursive (accessed Oct. 25, 2025).

‌