Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| LastName, FirstName MI. | Engr. Maria Rizette H. Sayo |

10/25/2025

# 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

**CODE SNIPPET:**

A screenshot of a computer program

AI-generated content may be incorrect.

BFS (Breadth-First Search) is the guaranteed shortest route it's like asking a search party to check every house one block at a time before moving to the next block. BFS is used when it’s need to find the closest person (or node) or the shortest path. DFS (Depth-First Search) is the deep dive it's like asking a search party to go as far down one specific alley as they can before they give up and try another one. DFS is used when checking for things like loops in the graph, performing a task ordering, or the thing we’re looking for is hidden way at the end of a long, deep path.

The main difference is memory usage, especially in large graphs. BFS can be a memory consuming method. It needs to keep all the nodes at the current exploration "level" in its queue simultaneously. If we have a super wide graph (lots of neighbors for every node), BFS can quickly run out of space. However, DFS is much more memory-friendly. It only needs to remember the nodes currently in the path it's exploring, making its space usage smaller, especially in wide graphs. This is why DFS is often the better choice when you're worried about memory limits.

The order is different because they use different tools. BFS uses a Queue (First-In, First-Out). It finds a node, puts all its neighbors in the queue, and then processes them in the exact order they were found. This results in the guaranteed level-by-level exploration. While DFS uses a Stack (Last-In, First-Out). It finds a node, pushes all its neighbors to the stack, and then immediately pulls the most recently added neighbor to explore it next. This creates the deep dive effect, prioritizing depth over breadth.

The recursive version of DFS is great because it's clean and simple, but it has a major flaw, the computer's call stack. Every time the recursive function calls itself, the computer saves data to the call stack. If the graph has a path that is incredibly long like 10,000 nodes deep, you'll get 10,000 nested function calls. This will exceed the programming language's recursion limit and trigger a Stack Overflow Error. The DFS iterative version avoids this completely by using a simple list or array as its stack in the main memory, which means its depth is limited only by the total memory available to your program, not by a small, fixed system limit.

# Conclusion

The key differences between BFS and DFS are their exploration method and resource usage. BFS uses a Queue for a level-by-level search, making it ideal for finding the shortest path, but it can be memory-consuming on wide graphs. DFS uses a Stack for a deep dive exploration, which is more memory-friendly and is more preferred. A crucial difference is that the recursive DFS risks a Stack Overflow Error on extremely deep paths.

**References**

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