Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Polestico,Paul Justine D. | 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

Question 1:

If i wanted to find topics related to that word when searching on google chrome(shortest path from the start node), i would prefer Breadth-First Search. If i needed to find a specific customer having hardware problems on their electronic devices on that particular day on customer service(How deep the graph is), i would prefer Depth-First Search.

Question 2:

The difference in space complexity between DFS and BFS is while they use the same time time complexity, they differ in while DFS provides depth-dependent memory potential, it can cause stack overflow if not careful. In BFS, while it has limited space, provides manageable memory in return,redcuing chances of stack overflow.

Question 3:

The difference in **traversal order** between DFS and BFS is that DFS explores **deeply** into one branch first before backtracking, following a **Last-In, First-Out (LIFO)** principle (like stacking plates). In contrast, BFS explores **broadly** by visiting all neighboring nodes in a level before advancing, following a **First-In, First-Out (FIFO)** principle (like a queue of people).

Question 4:

DFS recursive fails when it encounters a graph that is too **deep** (very high depth) because it causes a **Stack Overflow**. This is because the recursive version relies on the **Call Stack** (System Stack), which has a **fixed and limited size** for storing function calls.

In contrast, the DFS iterative version uses an **Explicit Stack** (usually a list or deque) stored in the **Heap Memory**, which is much **larger and more flexible**. While both have the same time complexity of O(V+E), the iterative approach avoids the strict size limit of the system's Call Stack, making it **less prone to Stack Overflow** on deep graphs.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1 Output of program

Conclusion

The two types of traversals discussed in this activity demonstrate the importance and usage of such traversals in different situations. Factors such as computer memory, algorithm, and computer capability affect the way programs run and process data efficiently and swiftly. It is implied that analyzation and understanding of traversals determine the overall performance of programs and the understanding of graphs in data structures.

**References**

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