UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Catahan, Joshua A.

Instructor:
Engr. Maria Rizette H. Sayo

11, 10, 2025

I. 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

II. 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
```

3. 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?

III. Results

When I first ran the program, it didn't show any output. I checked the code and saw that it only defined the `Graph`, `DFS`, and `BFS` functions but didn't actually call them. To fix this, I created a sample graph, added some vertices and edges, and then called the display, DFS, and BFS functions. After that, the program worked properly and showed the correct traversal outputs.

```
Graph Adjacency List:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B']
E: ['C']

DFS Recursive Traversal:
Visiting: A
Visiting: B
Visiting: D
Visiting: C
Visiting: E
['A', 'B', 'D', 'C', 'E']
```

```
DFS Iterative Traversal:
DFS Iterative Traversal:
Visiting: A
Visiting: B
Visiting: D
Visiting: C
Visiting: E
['A', 'B', 'D', 'C', 'E']

BFS Traversal:
BFS Traversal:
Visiting: A
Visiting: B
Visiting: C
Visiting: D
Visiting: E
['A', 'B', 'C', 'D', 'E']
```

Screenshot of the Program

Answers to the questions above:

1. I would prefer **DFS** when I need to explore a graph deeply before moving to other branches, such as in maze solving, topological sorting, or pathfinding problems where the goal might be far from the root. It's also useful when memory is limited because it doesn't store all nodes at once. On the other hand, I would choose **BFS** when I need to find the **shortest path** between nodes or explore all nearby nodes first, such as in social network connections or routing problems.

2. The **space complexity** of DFS is generally **O(h)**, where *h* is the height or depth of the graph, because it only needs to remember the current path. In contrast, **BFS** has a space complexity of **O(V)**, where *V* is the number of vertices, since it stores all the nodes at the current level in the queue. This makes BFS more memory-consuming, especially in wide graphs.
3. The **traversal order** of DFS and BFS is quite different. **DFS** goes as deep as possible into a branch before backtracking, giving a depth-based order. **BFS**, however, explores the graph level by level, visiting all immediate neighbors first before moving deeper. Because of this, BFS tends to find the shortest path in unweighted graphs, while DFS explores more thoroughly but not necessarily in the shortest way.
4. **DFS recursive** can fail when the graph or tree is too deep, because Python has a recursion limit that can cause a stack overflow error if there are too many recursive calls. In such cases, the **DFS iterative** version is more reliable since it uses an explicit stack instead of the function call stack. The iterative approach can handle larger and deeper graphs without crashing due to recursion limits.

IV. Conclusion

In this laboratory activity, I was able to understand and implement how graph traversal works using Depth-First Search (DFS) and Breadth-First Search (BFS). At first, the code didn't show any output because the functions were only defined and not executed, so I added sample data and function calls to make the program work properly. Through this, I observed how DFS explores paths deeply before backtracking, while BFS visits nodes level by level. I also learned that DFS usually uses less memory, while BFS consumes more due to its queue structure. Overall, this activity helped me clearly see the difference in their traversal order, space usage, and situations where each algorithm is more effective.

References

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