UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Asugas, Kenneth R.

Instructor:
Engr. Maria Rizette H. Sayo

10, 25, 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

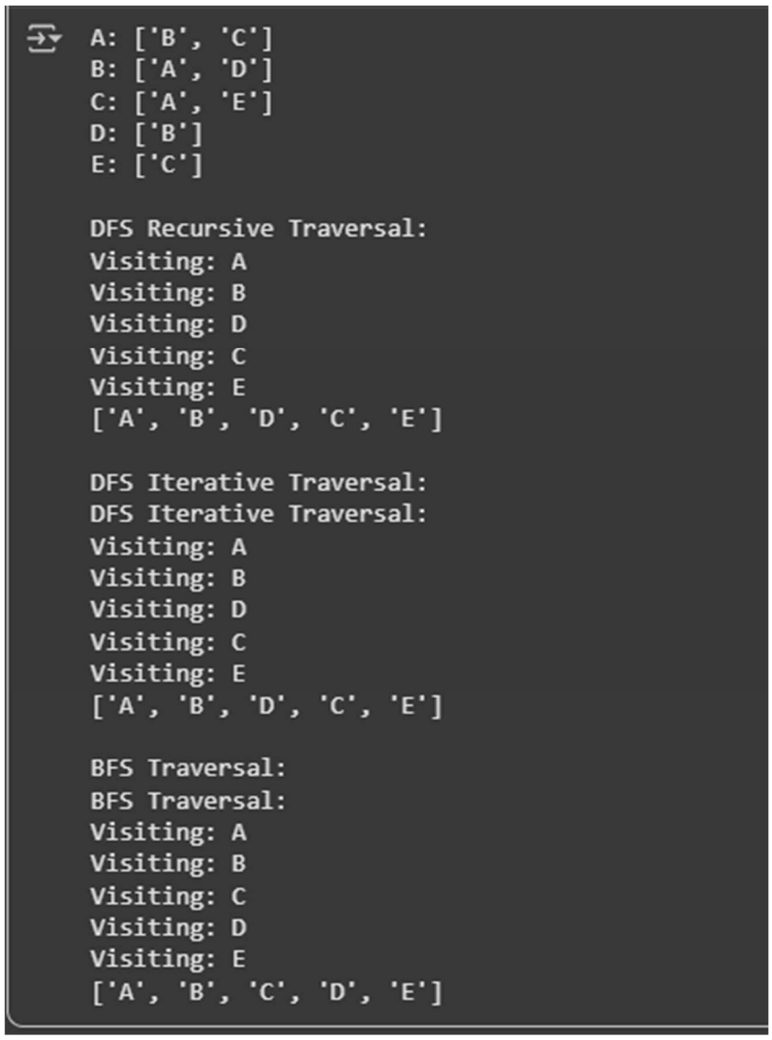

Figure 1 Screenshot of program

Answers:

1. DFS is preferred when the solution is likely to be found deep in the graph or when memory is limited since it uses less space. BFS is preferred when the shortest path or the nearest solution is needed, as it explores level by level.

2. DFS has a space complexity of O(V) due to the recursion stack or manual stack used. BFS also has O(V) space complexity, but in practice, it consumes more memory since it stores all neighbors at the current level in the queue.
3. DFS goes deep into one branch before proceeding to another, whereas BFS travels to all nodes of the current depth before venturing deeper. That's why DFS might traverse long paths initially, whereas BFS fans out uniformly from the source node.
4. DFS recursive can fail in deeply nested or extremely large graphs due to stack overflow from recursion stack limits. The iterative approach prevents this since it exploits an explicit stack that is manually managed on nodes.

IV. Conclusion

In this lab report, we looked at implementing and analyzing the Depth-First Search (DFS) and Breadth-First Search (BFS). These algorithms allowed us to understand the process of exploring a graph and how they handle data. We found that for situations in which one would want to explore a graph in depth, DFS is more efficient as it will 'go down' a path further, and may be more space efficient than BFS, which works across the level of a given graph to find the shortest units. While both DFS and BFS have the same time complexity of (V + E) they can vary widely in terms of space complexity based on the graph provided. Overall, this lab helped demonstrate how these algorithms work in practice, and provided a broader understanding of how these types of algorithms can be applied to real problems within the field of computer engineering.

References

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