



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Acebedo, Sebastian C.

Instructor:
Engr. Maria Rizette H. Sayo

October 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}')

```

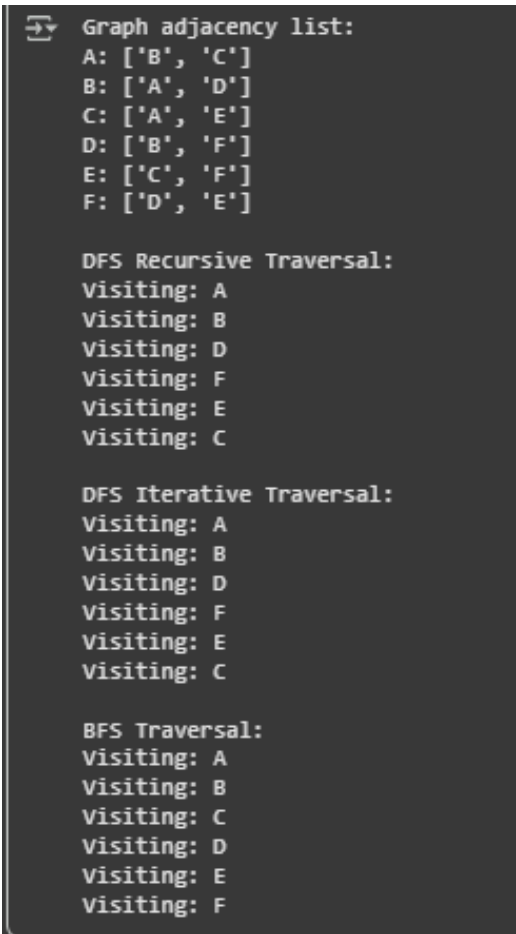
```
        for neighbor in graph.adj_list[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)

    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

A screenshot of a terminal window with a dark background and light-colored text. It displays the output of a graph traversal program. The output is organized into three sections: 'Graph adjacency list:', 'DFS Recursive Traversal:', and 'DFS Iterative Traversal:'. Each section shows the sequence of nodes visited during the traversal. The 'BFS Traversal:' section is also present at the bottom. The nodes are labeled A through F, and their neighbors are listed in the adjacency list. The traversal order for DFS recursive is A, B, D, F, E, C. For DFS iterative, it is A, B, D, F, E, C. For BFS, it is A, B, C, D, E, F.

```
Graph adjacency list:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'F']
E: ['C', 'F']
F: ['D', 'E']

DFS Recursive Traversal:
Visiting: A
Visiting: B
Visiting: D
Visiting: F
Visiting: E
Visiting: C

DFS Iterative Traversal:
Visiting: A
Visiting: B
Visiting: D
Visiting: F
Visiting: E
Visiting: C

BFS Traversal:
Visiting: A
Visiting: B
Visiting: C
Visiting: D
Visiting: E
Visiting: F
```

Figure 1
Screenshot of the Output

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

DFS is more applicable when we have to visit all the available paths or travel deep in a graph, such as for mazes or solving puzzles. BFS is more appropriate when we need to search for a shortest path or access a goal through the minimum number of steps.

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

They have the same space complexity of $O(V)$, but BFS tends to use more memory since it stores all nodes in the same level in the queue. DFS will use less memory as it tracks the current path using recursion or a stack.

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

DFS dives deeper first before it goes to another branch, whereas BFS examines all surrounding nodes first before diving deeper. Therefore, DFS takes one path at a time, and BFS goes level by level..

4. **When does DFS recursive fail compared to DFS iterative?**

DFS recursive can fail if the graph is too deep because it would lead to a stack overflow due to too many recursive calls. The iterative version does not have this issue as it employs a stack that we handle manually.

IV. Conclusion

Through this lab exercise, I discovered the operations of Depth-First Search (DFS) and Breadth-First Search (BFS) and how they are different in traversal and performance. DFS searches a graph by diving deep down one path before it backtracks, whereas BFS traverses all the nodes level by level. Both have the same $O(V + E)$ time complexity, but BFS tends to take more space because it keeps a lot of nodes in the queue, whereas DFS takes less space as it only keeps the current path. I also noticed that recursive DFS can cause stack overflow in extremely deep graphs, whereas the iterative one does not. Overall, this exercise gave me an idea of when to utilize DFS for exploring deeply and BFS for determining the shortest path based on the nature of the problem.

References

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