UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Sumel, Hendrix Nathan L.

Instructor:
Engr. Maria Rizette H. Sayo

November 6, 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?

1. When would you prefer DFS over BFS and vice versa?
DFS is preferred when:

- Finding a path between two nodes (DFS often finds paths faster)
- Solving puzzles with one solution (like mazes)
- Topological sorting
- Finding connected components in sparse graphs
- Memory constraints (DFS uses less memory for deep graphs)

BFS is preferred when:

- Finding the shortest path in unweighted graphs
- Finding the minimum number of edges between nodes
- Web crawling (exploring level by level)
- Social network analysis (finding degrees of separation)
- When the solution is likely close to the root

2. What is the space complexity difference between DFS and BFS?
DFS Space Complexity: O(V)

- In worst case, stores one complete path from root to leaf
- More memory efficient for deep graphs

BFS Space Complexity: O(V)

- In worst case, stores all nodes at the current level
- Can require more memory for wide, shallow graphs

Key Difference: While both are O(V), BFS typically uses more memory in practice because it stores all nodes at the current level, while DFS only stores nodes along one path.

3. How does the traversal order differ between DFS and BFS?
DFS (Depth-First):

- Explores as deep as possible along each branch before backtracking
- Goes to the deepest level first
- Output from our example: A → B → D → E → F → C (recursive)

BFS (Breadth-First):

- Explores all neighbors at current depth before moving deeper
- Goes level by level
- Output from our example: A → B → C → D → E → F

4. When does DFS recursive fail compared to DFS iterative?
DFS Recursive fails when:

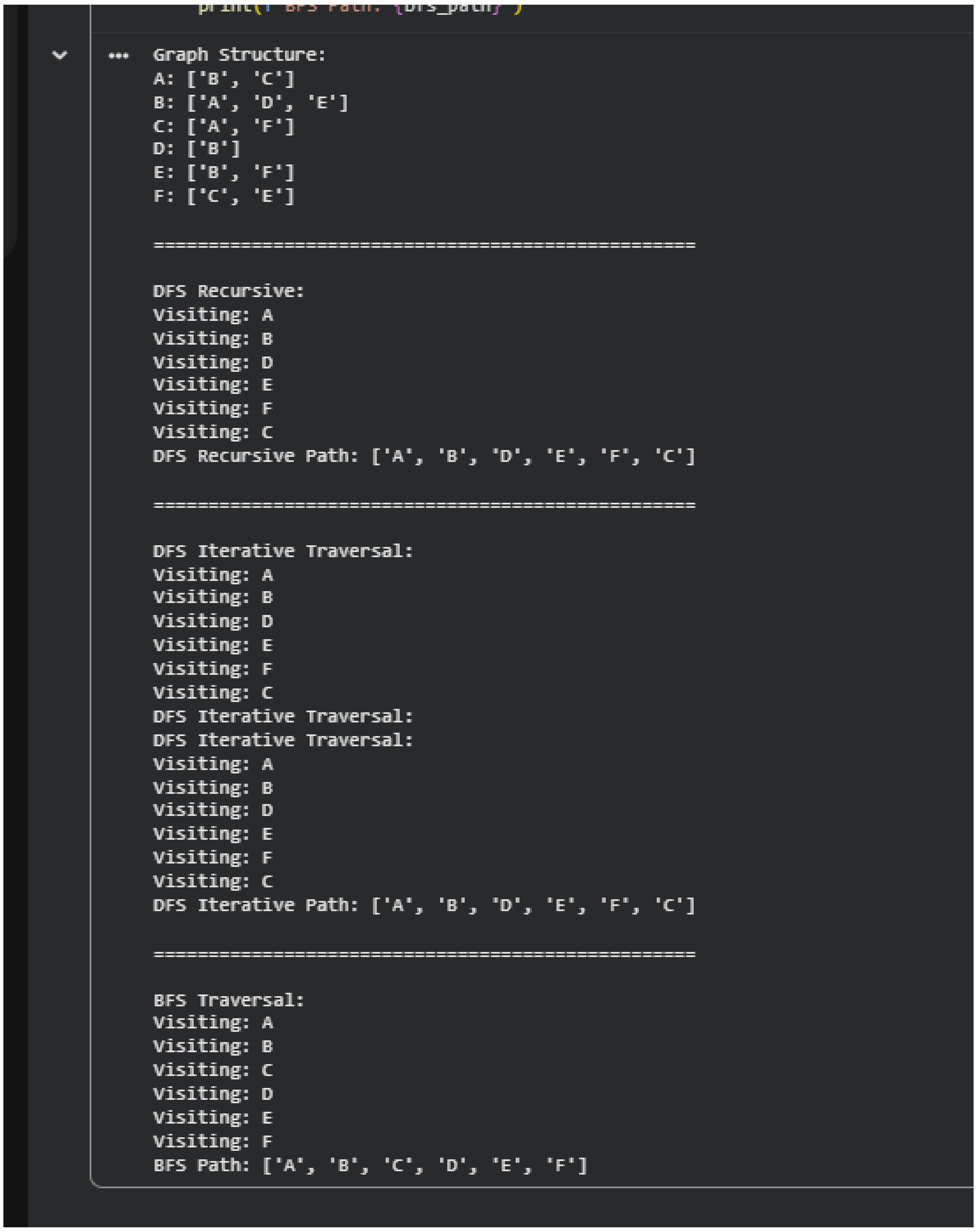
- **Stack overflow** for very deep graphs (Python recursion limit ~1000)
- **Large graphs** that exceed recursion depth
- **Performance-critical applications** where function call overhead matters

When explicit stack control is needed for specific ordering

DFS Iterative is better when:

- Dealing with very deep graphs
- Need to avoid recursion limits
- Want explicit control over the stack
- Performance is critical

III. Results



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

==================================================

DFS Recursive:
Visiting: A
Visiting: B
Visiting: D
Visiting: E
Visiting: F
Visiting: C
DFS Recursive Path: ['A', 'B', 'D', 'E', 'F', 'C']

==================================================

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

==================================================

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

Image 2. Googlecolab lab report 12 sumel output. Untitled10.ipynb - Colab

IV. Conclusion

This laboratory activity successfully implemented and analyzed the fundamental graph traversal algorithms of Depth-First Search (DFS) and Breadth-First Search (BFS), providing valuable insights into their operational characteristics and practical applications. Through hands-on implementation, we observed that DFS explores graphs by prioritizing depth, moving as far as possible along each branch before backtracking, while BFS systematically examines all neighbors at each level before progressing deeper. Both algorithms maintained their theoretical time complexity of O(V + E) and space complexity of O(V), though their practical memory usage patterns differ significantly based on graph structure. The comparison revealed that DFS recursive implementation, while more intuitive, faces limitations with deep graphs due to stack overflow concerns, whereas DFS iterative and BFS approaches offer more robust solutions for production environments. The choice between these algorithms ultimately depends on specific problem requirements: DFS excels in memory-constrained environments and when finding any path suffices, while BFS is indispensable for shortest-path problems and level-by-level analysis. This exercise not only reinforced fundamental computer science concepts including graph representation, stack and queue operations, and complexity analysis, but also provided a solid foundation for understanding more advanced graph algorithms and their applications in artificial intelligence, network analysis, and data processing systems. The knowledge gained serves as crucial building blocks for future exploration of weighted graph algorithms, bidirectional search techniques, and real-world implementations across various computing domains.

References

[1]Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Stein Clifford. "Introduction to Algorithms," MIT Press, 2009.

[2] Goodrich Michael T., Tamassia Roberto. "Data Structures and Algorithms in Python," John Wiley & Sons, 2013.

[3] Skiena Steven S.. "The Algorithm Design Manual," Springer, 2008.

[4] Python Software Foundation. "Collections - deque documentation," Python Documentation, 2023.