UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Ampong, J-kevin L.

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

1.

Depth-First Search (DFS) is generally preferred when you want to move as far as possible along each branch before backtracking. It's particularly useful in uses like topological sorting, puzzle or maze solving, and path searching where the solution is likely to be deep in the graph. DFS also uses less memory for traversal on sparse graphs or graphs with long, thin structures. BFS is preferred, though, when you want to find the shortest path in an unweighted graph or visit all nodes level by level. It’s widely used in applications like finding the shortest route in navigation systems, peer-to-peer networks, or social network friend recommendations, where the goal is to find the minimum number of edges between nodes.

2.

The main difference between DFS and BFS lies in their space requirements. BFS typically requires more memory since it stores all nodes at the current level before moving on to the next. In the worst case, for a graph with a large branching factor, BFS will take space proportional to (O(V)), where (V) is the number of vertices, or more precisely (O(b^d)), where (b) is the branching factor and (d) is the depth of the shallowest goal node. On the other hand, DFS usually has a lower space complexity of (O(V)) or (O(bm)), where (m) is the maximum depth of the graph. This is due to the fact that DFS only needs to maintain the current path from the root to the leaf node, along with the unvisited siblings of all the nodes along the path.

3.

DFS and BFS differ in how they traverse the graph. DFS visits nodes by going deep along one branch before backtracking to visit other branches. This results in a visitation order that follows one path all the way down before moving to another, creating a depth-first traversal. BFS, on the other hand, visits nodes level by level — it visits all of a node's immediate neighbors before moving on to the children. Therefore, BFS's traversal order extends from the source node in a manner that all nodes at distance 1 are traversed before those at distance 2, and so on. This difference in ordering makes BFS a strong candidate for shortest-path queries and DFS a strong candidate for structure discovery or connectivity testing.

4.

DFS recursive traversal can fail in the scenario of very deep or gigantic graphs due to stack overflow issues. A new frame on the system call stack is created with each recursive call, and when the recursion depth is greater than the system's maximum tolerance limit, the program aborts. This often happens in graphs with thousands or millions of nodes connected in a gigantic chain-like structure. On the other hand, DFS iterative averts this issue by utilizing an explicit stack data structure to manage traversal, which is more safely capable of managing larger depths since it does not utilize the system's finite recursion stack. Therefore, for deep or complex graphs, the iterative version of DFS is more robust and less prone to memory-based failures.

IV. Conclusion

As a conclusion, Depth-First Search (DFS) and Breadth-First Search (BFS) are two basic graph traversal algorithms that are useful for certain applications depending on the nature of the problem. DFS is better when it is about searching deep in a graph and is therefore suitable for problems like pathfinding, solving mazes, or detecting cycles where one has to search all of them. It is also memory-efficient as it requires less memory compared to BFS. Recursive DFS, however, will crash in very deep graphs due to stack overflow, while the iterative one is a more secure choice. BFS, however, is better for searching the shortest path in unweighted graphs and visits nodes level by level in a systematic manner, thus being handy for network analysis and routing. Finally, a decision between DFS and BFS would only be based on whether the problem requires the search of depth or shortest-path finding in priority, and that the knowledge of both algorithms' properties and trade-offs is essential.

References

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