UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Bron, Jhustine A.

Instructor:
Engr. Maria Rizette H. Sayo

October 30, 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

Answers:

1. I would use BFS (Breadth-First Search) when I need to find the *shortest path* in an unweighted graph since BFS explores level by level, which means it visits the closest nodes first. This makes it perfect for things like shortest path problems or situations where the goal is likely near the starting point. On the other hand, I'd prefer DFS (Depth-First Search) when I want to explore deeper paths or check the structure of the entire graph, such as when detecting cycles or finding connected components. DFS is also useful when the graph is wide because it doesn't have to keep all nodes in memory like BFS does. So basically, if I want to find something quickly and close by, I'll use BFS; if I want to explore deeply or analyze structure, I'll use DFS.
2. Both DFS and BFS have the same time complexity of $O(V + E)$, but their space usage is a bit different. BFS uses a queue to store all the nodes at the current level, so if the graph is wide, the queue can become very large, making its space complexity $O(V)$. DFS also has a space complexity of $O(V)$, but it depends more on the depth of the graph rather than its width. When I use the recursive version of DFS, it relies on the call stack, which can get really deep and even cause a stack overflow if the graph is long. The iterative version of DFS avoids this by using a manual stack that I can control. In general, BFS tends to use more memory for wide graphs, while DFS might struggle with deep ones.
3. The main difference I notice between BFS and DFS is how they explore nodes. BFS visits nodes level by level, it starts from the root, then visits all nodes that are one edge away, then two edges away, and so on. DFS, on the other hand, goes as deep as possible down one path before backtracking to explore other branches. This means BFS explores outward in layers, while DFS dives down like a tunnel. Because of that, BFS will always find the shortest path in an unweighted

graph first, but DFS might reach the destination sooner through a longer path. So, if I imagine exploring a maze, BFS would check all nearby turns evenly, while DFS would pick one path and follow it until the end before trying the others.

4. In my experience, recursive DFS can fail when the graph is too deep because it relies on the system's call stack to keep track of recursive calls. If the recursion goes too far, it can hit the recursion limit and crash due to a stack overflow. The iterative version avoids this issue because it uses my own stack structure, which gives me more control and lets me handle deeper graphs safely. Recursive DFS is simpler to write and works fine for smaller graphs, but for larger or unpredictable ones, I'd rather use the iterative version to avoid stack-related errors. Both versions can loop forever if I forget to mark nodes as visited, but that's more of a coding mistake than a recursion issue.

IV. Conclusion

In conclusion, both DFS and BFS are essential graph traversal algorithms, and the choice between them really depends on the problem I'm trying to solve. I've learned that BFS is more suitable when I need to find the shortest path or explore nodes closer to the start first, while DFS is better when I want to go deep into the graph or check for structures like cycles and connections. Even though both have similar time complexity, their space usage and traversal behavior are quite different. I also realized that recursive DFS can run into stack issues with deep graphs, so the iterative version is often safer for large ones. Overall, understanding when and how to use each algorithm helps me apply the right one efficiently in different situations.

References

GeeksforGeeks, “Difference between BFS and DFS,” *GeeksforGeeks*, Jul. 11, 2025. https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

“Difference between BFS and DFS.” https://www.tutorialspoint.com/difference-between-bfs-and-dfs

Codecademy, “Breadth-First Search vs Depth-First Search: Key Differences,” *Codecademy*. https://www.codecademy.com/article/bfs-vs-dfs