



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Directo, Hannah Thea B.

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. I would prefer to use DFS when I need to explore a graph deeply before moving to other paths. It's useful for things like maze solving, checking possible routes, or exploring deep structures. It also uses less memory, which makes it good for smaller or deeper graphs. I would use BFS when I need to find the shortest path or explore nodes level by level. This is helpful in cases like route finding or network searches. In short, I'd use DFS to go deep and BFS to go wide.

2. DFS takes up less memory because it only keeps track of the path it's currently exploring. BFS, on the other hand, uses more memory since it stores all the nodes at the current level. So, DFS is more memory-efficient, while BFS can take up more space, especially in large or wide graphs.

3. DFS explores one path completely before moving to another, meaning it goes deep first. BFS explores all nearby nodes first before moving on to the next level, so it goes level by level. DFS dives deep, while BFS spreads out evenly.

4. The recursive DFS can fail when the graph is too large or too deep because it depends on the system's call stack, which has a limit. Once that limit is reached, it causes a recursion error. The iterative DFS avoids this problem since it uses its own stack, making it work better for large graphs.

IV. Conclusion

The activity showed how DFS and BFS explore graphs in different ways. DFS goes deep first and uses less memory, while BFS goes level by level and is better for finding the shortest path. Both methods are important in solving real-world problems like routing, network analysis, and pathfinding.