UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Disomnong, Jalilah M.

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

```
[3]
#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}")
```

Figure 2 Source code of Graph link: LAB 12 - Colab

```
✓ 0s
#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
```

Figure 1 Source code of DFS link: LAB 12 - Colab

```
[5]
✓ 0s
#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
```

Figure 3 Source code of BFS link: LAB 12 - Colab

In question 1, I would use BFS when I need to find the shortest path or explore nodes close to the starting point since it moves level by level. On the other hand, I will use DFS when I want to explore deeper paths or when the solution might be far from the start, like in maze solving or backtracking problems. So, BFS is better for quick, shallow searches, while DFS is better for deep or exhaustive ones.

In question 2, The space complexity of BFS is generally higher than that of DFS because BFS uses a queue to store all the nodes at the current level before moving on to the next. This means that if the graph is very wide, the queue can hold a large number of nodes, leading to a space complexity of O(V), where V is the number of vertices. DFS, on the other hand, uses a stack to keep track of the nodes it visits, either an explicit stack in the iterative version or the call stack in the recursive version. Its space complexity is O(h), where h is the maximum depth of the graph, and at worst O(V) if the graph is very deep. Therefore, DFS usually consumes less memory than BFS, especially when the graph is wide but shallow. (GeeksforGeeks, 2025a)

In question 3, BFS and DFS differ fundamentally in the order in which they explore nodes. BFS explores the graph level by level, visiting all immediate neighbors of a node before moving on to their neighbors. This means that it traverses the graph in a breadthwise manner, resulting in a traversal order such as A → B → C → D → E → . DFS, in contrast, explores as far down a path as possible before backtracking, creating a depth-wise traversal pattern. Its order of visitation might look like A → B → D → E → C → F, depending on the structure of the graph. In simple

terms, BFS explores the graph horizontally (level by level), while DFS explores it vertically (deep into branches). (GeeksforGeeks, 2025a)

In question 4, Recursive DFS can fail on very large or deep graphs because it depends on the system's call stack, which has a limited depth. If the recursion goes too deep, it can cause a stack overflow error. Iterative DFS avoids this by using its own stack, making it more reliable for large or deeply nested graphs. (GeeksforGeeks, 2025a)

IV. Conclusion

In this lab, we explored and practiced using Depth-First Search (DFS) and Breadth-First Search (BFS) on graphs. We found that BFS is effective for finding the shortest path and exploring nearby nodes first, while DFS is better suited for going deep or checking all possible paths. We also observed differences in traversal order and memory usage, and noted that recursive DFS can struggle with very large graphs, whereas iterative DFS handles them more reliably. Therefore, this activity reinforced our understanding of how these algorithms work and when to apply each one in solving practical problems.

References

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

[2] GeeksforGeeks. (2025a, July 11). *Difference between BFS and DFS*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

[3] Google. (2024). Google Colaboratory. Retrieved April 18, 2024, from https://colab.research.google.com/