UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 12

Graph Searching Algorithm

Submitted by:
Palmes, Lewis Clark 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

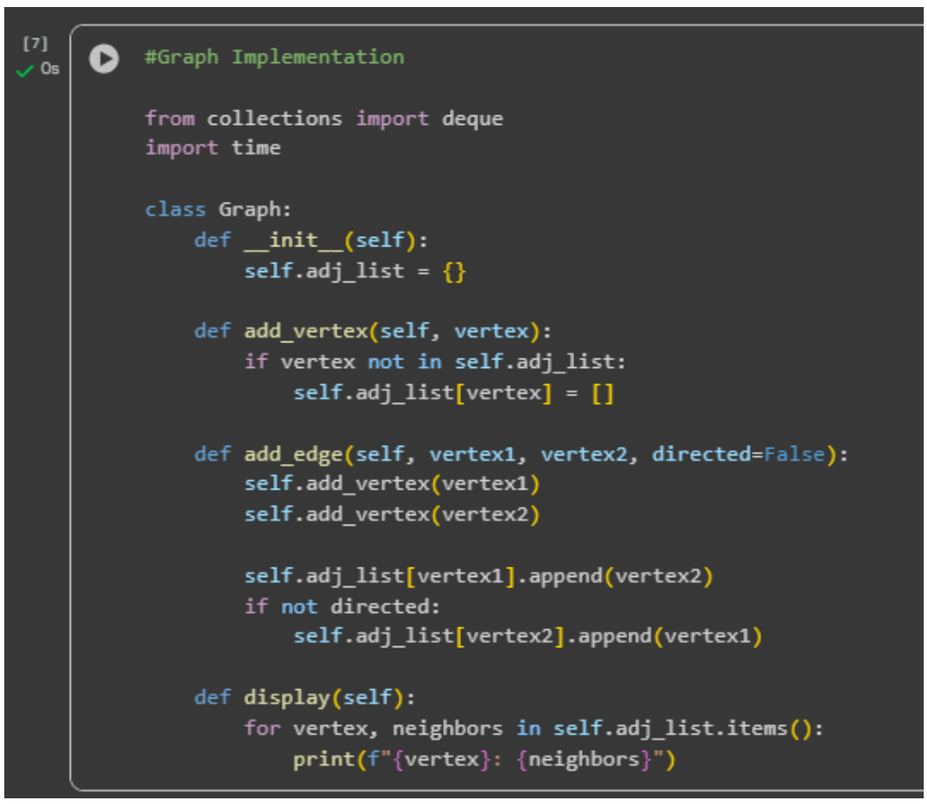
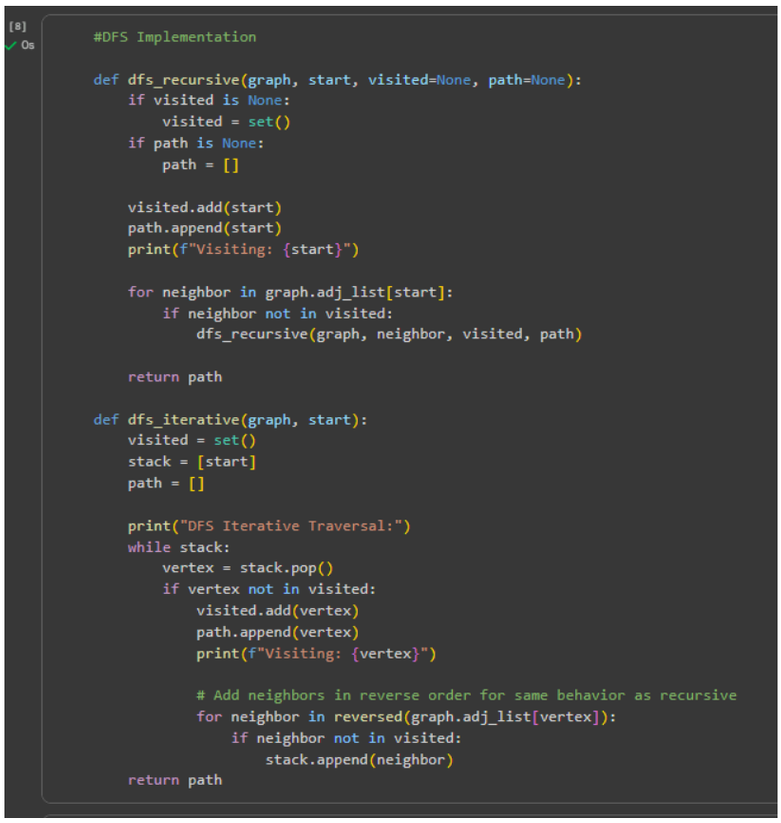
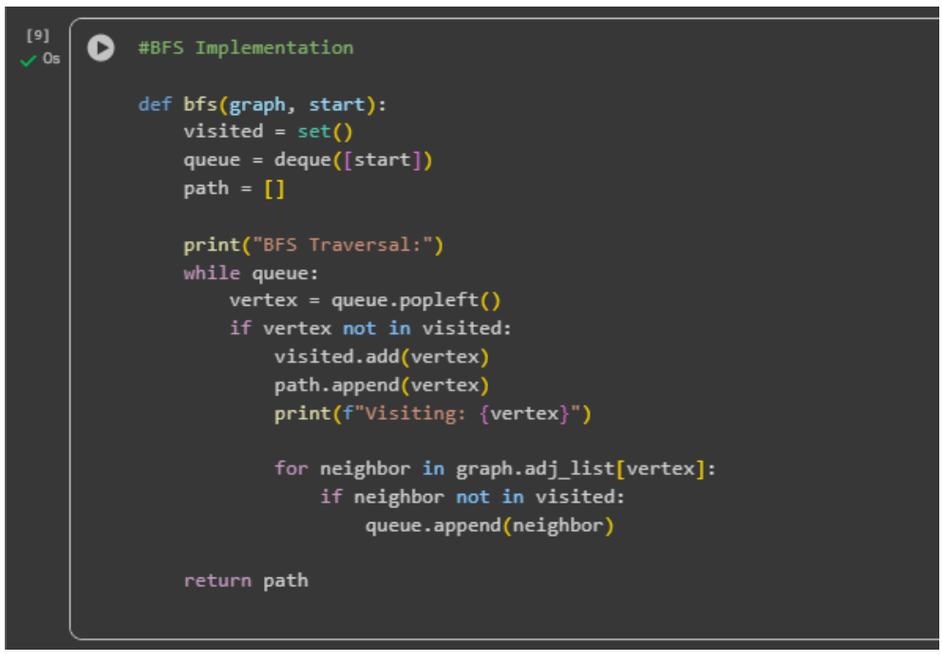


Figure 1 Graph Implementation program



```
[8]
✓ 0s
    #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 2 DFS Implementation



```
[9]
✓ 0s
    #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 BFS Implementation

1. When would you prefer DFS over BFS and vice versa?

   The choice between DFS and BFS depends on the problem structure. DFS is preferred for tasks requiring deep exploration, such as path existence checks, cycle detection, and topological sorting, as it follows each branch to completion before backtracking. BFS excels at finding shortest paths in unweighted graphs and level-order traversal, systematically exploring all neighbors at each depth before progressing deeper.

2. What is the space complexity difference between DFS and BFS?

   Both algorithms have O(V) space complexity in the worst case, but their memory usage patterns differ. DFS maintains a stack of nodes along the current path, making it more memory-efficient for deep, narrow graphs. BFS stores all nodes at the current depth level in a queue, often requiring more memory for broad, dense graphs.

3. How does the traversal order differ between DFS and BFS?

   The traversal order fundamentally differs due to their data structures. DFS uses a stack to pursue vertical depth, rapidly moving away from the start node along single branches. BFS employs a queue for horizontal exploration, visiting all immediate neighbors before progressing to subsequent layers, ensuring systematic level-by-level coverage.

4. When does DFS recursive fail compared to DFS iterative?

   Recursive DFS fails when graph depth exceeds language recursion limits, causing stack overflow errors on deep graphs. Iterative DFS manages its own stack structure, avoiding call stack limitations and providing reliable performance for graphs of any depth, making it the robust choice for production systems.

IV. Conclusion

This activity helped me better understand how DFS and BFS actually work in practice by implementing and testing them on a graph. I was able to see how DFS explores deeper paths first, while BFS visits nodes level by level. Through comparing them, I learned when each algorithm is more useful and how their space usage differs based on graph structure. Overall, this laboratory strengthened my grasp of graph traversal and gave me hands-on experience applying these concepts in code.

References

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

[2] GeeksforGeeks, “Depth first search or DFS for a graph,” *GeeksforGeeks*, Oct. 03, 2025. https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/

[3] “Google Colab.” https://colab.research.google.com/drive/1iY_Xi_qckf_pk7idLTrkalEQPx_OBHph#scrollTo=Q7Wt8857LMO6

[4] GeeksforGeeks, “Breadth first search or BFS for a graph,” *GeeksforGeeks*, Oct. 21, 2025. https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/

[5] Lewis-Clark-Palmes, “CPE-201L-DSA-2-A/LAB-12.ipynb at main · Lewis-Clark-Palmes/CPE-201L-DSA-2-A,” *GitHub*. https://github.com/Lewis-Clark-Palmes/CPE-201L-DSA-2-A/blob/main/LAB-12.ipynb