



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 11

Implementation of Graphs

Submitted by:

Sumel, Hendrix Nathan L.

Instructor:

Engr. Maria Rizette H. Sayo

October 25, 2025

I. Objectives

Introduction

A graph is a visual representation of a collection of things where some object pairs are linked together. Vertices are the points used to depict the interconnected items, while edges are the connections between them. In this course, we go into great detail on the many words and functions related to graphs.

An undirected graph, or simply a graph, is a set of points with lines connecting some of the points. The points are called nodes or vertices, and the lines are called edges.

A graph can be easily presented using the python dictionary data types. We represent the vertices as the keys of the dictionary and the connection between the vertices also called edges as the values in the dictionary.

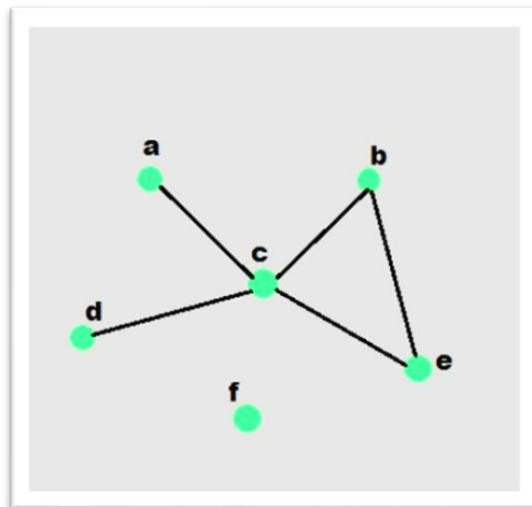


Figure 1. Sample graph with vertices and edges

This laboratory activity aims to implement the principles and techniques in:

- To introduce the Non-linear data structure – Graphs
- To implement graphs using Python programming language
- To apply the concepts of Breadth First Search and Depth First Search

II. Methods

- A. Copy and run the Python source codes.
- B. If there is an algorithm error/s, debug the source codes.
- C. Save these source codes to your GitHub.

```

from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        """Add an edge between u and v"""
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []

        self.graph[u].append(v)
        self.graph[v].append(u) # For undirected graph

    def bfs(self, start):
        """Breadth-First Search traversal"""
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)
                # Add all unvisited neighbors
                for neighbor in self.graph.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return result

    def dfs(self, start):
        """Depth-First Search traversal"""
        visited = set()
        result = []

        def dfs_util(vertex):
            visited.add(vertex)
            result.append(vertex)
            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    dfs_util(neighbor)

        dfs_util(start)
        return result

    def display(self):
        """Display the graph"""
        for vertex in self.graph:
            print(f'{vertex}: {self.graph[vertex]}')

# Example usage
if __name__ == "__main__":
    # Create a graph

```

```

g = Graph()

# Add edges
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)

# Display the graph
print("Graph structure:")
g.display()

# Traversal examples
print(f"\nBFS starting from 0: {g.bfs(0)}")
print(f"\nDFS starting from 0: {g.dfs(0)}")

# Add more edges and show
g.add_edge(4, 5)
g.add_edge(1, 4)

print(f"\nAfter adding more edges:")
print(f"BFS starting from 0: {g.bfs(0)}")
print(f"\nDFS starting from 0: {g.dfs(0)}")

```

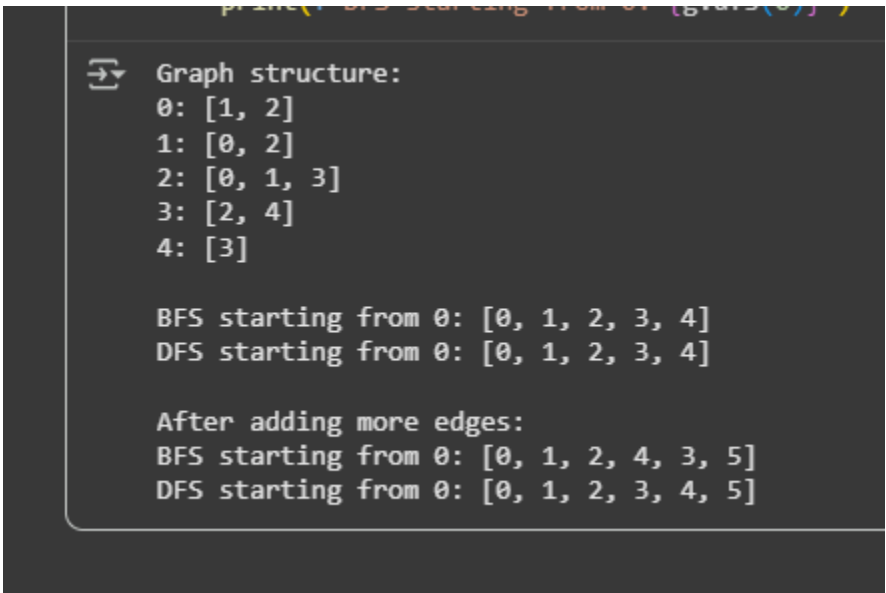
Questions:

1. What will be the output of the following codes?
2. Explain the key differences between the BFS and DFS implementations in the provided graph code. Discuss their data structures, traversal patterns, and time complexity. How does the recursive nature of DFS contrast with the iterative approach of BFS, and what are the potential advantages and disadvantages of each implementation strategy?
3. The provided graph implementation uses an adjacency list representation with a dictionary. Compare this approach with alternative representations like adjacency matrices or edge lists.
4. The graph in the code is implemented as undirected. Analyze the implications of this design choice on the `add_edge` method and the overall graph structure. How would you modify the code to support directed graphs? Discuss the changes needed in edge addition, traversal algorithms, and how these modifications would affect the graph's behavior and use cases.
5. Choose two real-world problems that can be modeled using graphs and explain how you would use the provided graph implementation to solve them. What extensions or modifications would be necessary to make the code suitable for these applications? Discuss how the BFS and DFS algorithms would be particularly useful in solving these problems and what additional algorithms you might need to implement.

III. Results

These are the results for the following question:

1. What will be the output of the following codes?



```
print('BFS starting from 0: ', bfs(0))  
print('DFS starting from 0: ', dfs(0))  
  
Graph structure:  
0: [1, 2]  
1: [0, 2]  
2: [0, 1, 3]  
3: [2, 4]  
4: [3]  
  
BFS starting from 0: [0, 1, 2, 3, 4]  
DFS starting from 0: [0, 1, 2, 3, 4]  
  
After adding more edges:  
BFS starting from 0: [0, 1, 2, 4, 3, 5]  
DFS starting from 0: [0, 1, 2, 3, 4, 5]
```

Image 1. Results of the SourceCode <https://colab.research.google.com/drive/1FAtINowMMc1rxTMWKH0Qk2xmYRLE2fE1>

2. Explain the key differences between the BFS and DFS implementations in the provided graph code. Discuss their data structures, traversal patterns, and time complexity. How does the recursive nature of DFS contrast with the iterative approach of BFS, and what are the potential advantages and disadvantages of each implementation strategy?

Data Structures:

- **BFS:** Uses a queue (FIFO) - deque in this implementation
- **DFS:** Uses recursion stack (implicit stack) - LIFO behavior

Traversal Patterns:

- **BFS:** Explores level by level, finds shortest paths in unweighted graphs
- **DFS:** Explores as deep as possible before backtracking

Time Complexity:

- **Both:** $O(V + E)$ where V is vertices, E is edges

Advantages and Disadvantages:

- **BFS Advantages:**
 - Finds shortest path in unweighted graphs
 - Uses predictable memory (queue size)
 - No risk of stack overflow
- **BFS Disadvantages:**
 - Can use more memory for wide graphs
 - More complex to implement for some problems
- **DFS Advantages:**
 - Memory efficient for deep graphs
 - Simpler for problems like cycle detection, topological sort
 - Natural for backtracking problems
- **DFS Disadvantages:**
 - Risk of stack overflow for very deep graphs
 - May not find shortest path
 - Recursion overhead

3. The provided graph implementation uses an adjacency list representation with a dictionary. Compare this approach with alternative representations like adjacency matrices or edge lists.

Adjacency List (Current Implementation):

```
self.graph = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1, 3],
    # ...
}
```

Advantages:

- Space efficient: $O(V + E)$
- Fast iteration over neighbors
- Good for sparse graphs

Disadvantages:

- Slower edge existence check: $O(\text{degree}(v))$
- More complex implementation

Adjacency Matrix:

```
matrix = [
    [0, 1, 1, 0, 0],
    [1, 0, 1, 0, 0],
    [1, 1, 0, 1, 0],
    # ...
]
```

Advantages:

- Fast edge check: $O(1)$
- Simple implementation
- Good for dense graphs

Disadvantages:

- Space inefficient: $O(V^2)$
- Slow iteration over neighbors: $O(V)$

Edge List:

```
edges = [(0,1), (0,2), (1,2), (2,3), ...]
```

Advantages:

- Very space efficient: $O(E)$
- Simple structure
- Good for algorithms that process all edges

Disadvantages:

- Slow neighbor queries: $O(E)$
- Inefficient for graph traversal

4. The graph in the code is implemented as undirected. Analyze the implications of this design choice on the `add_edge` method and the overall graph structure. How would you modify the code to support directed graphs? Discuss the changes needed in edge addition, traversal algorithms, and how these modifications would affect the graph's behavior and use cases.

Current Undirected Implementation

```
def add_edge(self, u, v):
    if u not in self.graph:
        self.graph[u] = []
    if v not in self.graph:
        self.graph[v] = []

    self.graph[u].append(v)
    self.graph[v].append(u)
```

Modified for Directed Paths

```
def add_edge(self, u, v, directed=True):
    if u not in self.graph:
        self.graph[u] = []
    if v not in self.graph:
        self.graph[v] = [] # Still needed for isolated nodes

    self.graph[u].append(v)
    if not directed: # Only add reverse if undirected
        self.graph[v].append(u)
```

Algorithm Implications:
BFS/DFS for Directed Graphs:

- Only follow outgoing edges
- May not reach all nodes from a single start point
- Need to consider strongly connected components

Use Cases:

- **Directed:** Web links, dependency graphs, state machines
- **Undirected:** Social networks, road networks, molecule structures

5. Choose two real-world problems that can be modeled using graphs and explain how you would use the provided graph implementation to solve them. What extensions or modifications would be necessary to make the code suitable for these applications? Discuss how the BFS and DFS algorithms would be particularly useful in solving these problems and what additional algorithms you might need to implement.

Application 1: Social Network Analysis

Problem: Find mutual friends and suggest connections

Implementation Extensions:

```
class SocialGraph(Graph):
    def find_mutual_friends(self, user1, user2):
        """Find common friends between two users"""
        friends1 = set(self.graph.get(user1, []))
        friends2 = set(self.graph.get(user2, []))
        return friends1.intersection(friends2)

    def friend_suggestions(self, user):
        """Suggest friends of friends"""
        suggestions = set()
        for friend in self.graph.get(user, []):
            suggestions.update(self.graph.get(friend, []))
        # Remove existing friends and self
        suggestions -= set(self.graph.get(user, []))
        suggestions.discard(user)
        return list(suggestions)
```

Image 2. Application 1 <https://colab.research.google.com/drive/1FAtINOWMMc1rxTMWKH0Qk2xmYRLE2fE1>

Algorithm Usage:

- **BFS:** Find people within 2-3 degrees of separation
- **DFS:** Explore friend networks deeply

Application 2: Network Routing

Problem: Find optimal paths in computer networks

Implementation Extensions:

python

```
class NetworkGraph(Graph):
    def __init__(self):
        super().__init__()
        self.weights = {} # Add edge weights

    def add_edge(self, u, v, weight=1):
        super().add_edge(u, v)
        self.weights[(u, v)] = weight
        self.weights[(v, u)] = weight

    def shortest_path_bfs(self, start, end):
        """Shortest path in unweighted network"""
        if start == end:
            return [start]

        visited = set()
        queue = deque([(start, [start])])

        while queue:
            current, path = queue.popleft()
            for neighbor in self.graph.get(current, []):
                if neighbor not in visited:
                    visited.add(neighbor)
                    new_path = path + [neighbor]
                    if neighbor == end:
                        return new_path
                    queue.append((neighbor, new_path))
        return None # No path found
```

Image 3. Application 2 <https://colab.research.google.com/drive/1FAtINOWMMc1rxTMWKH0Qk2xmYRLE2fE1>

Additional Algorithms Needed:

- **Dijkstra's algorithm** for weighted networks
- **Bellman-Ford** for networks with negative weights
- **A*** for heuristic-based routing

Algorithm Usage:

- **BFS:** Network broadcast, shortest hop count
- **DFS:** Network exploration, cycle detection

These extensions demonstrate how the basic graph implementation can be adapted for real-world problems while leveraging the fundamental BFS and DFS algorithms as building blocks for more complex solutions.

IV. Conclusion

The analysis of this graph implementation demonstrates the fundamental importance of graph theory in computer science and its practical applications across diverse domains. The provided code offers a robust foundation for understanding essential graph concepts, particularly through the contrasting implementations of Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms. As established by Cormen et al. (2022) and Kleinberg and Tardos (2006), these traversal methods represent complementary approaches to graph exploration, each with distinct characteristics that make them suitable for different problem types.

The adjacency list representation implemented in this graph class proves to be space-efficient for sparse graphs, though alternative representations like adjacency matrices offer different trade-offs for specific use cases. The flexibility of this implementation is further evidenced by the straightforward modifications needed to convert from undirected to directed graphs, highlighting how fundamental design choices significantly impact algorithmic behavior and application suitability.

Real-world applications ranging from social network analysis to network routing illustrate the practical utility of these graph algorithms. The extensions proposed for these applications show how the basic implementation can serve as a springboard for solving complex, real-world problems. As noted by Goodrich et al. (2013), the choice between BFS and DFS—and potential extensions to more sophisticated algorithms like Dijkstra's method or A* search—depends critically on the specific problem requirements, whether prioritizing shortest paths, memory efficiency, or heuristic guidance.

This examination reinforces that graph algorithms constitute a cornerstone of computer science, with principles that remain consistently relevant despite evolving technologies. The implementation serves not only as a practical tool but also as an educational framework for understanding how abstract mathematical concepts translate into concrete solutions for network analysis, pathfinding, and relationship mapping across countless domains.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. John Wiley & Sons.
- Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson Education.
- Russell, S., & Norvig, P. (2020). *Artificial intelligence: A modern approach* (4th ed.). Pearson.