MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of a Graph and its Depth First and Breadth First Traversals

Assignment No. 5

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

March 28, 2023

Contents

1 Objectives

1. *To study data structure Graph and its representation using adjacency list*
2. *To study and implement recursive Depth First Traversal and use of stack data*
3. *structure for recursive Depth First Traversal*
4. *To study and implement Breadth First Traversal*
5. *To study how graph can be used to model real world problems*

2 Problem Statement

Consider a friend's network on Facebook social web site. Model it as a graph to represent each node as a user and a link to represent the friend relationship between them using adjacency list representation and perform DFS and BFS traversals.

3 Theory

3.1 Definition of Graph

A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, a Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

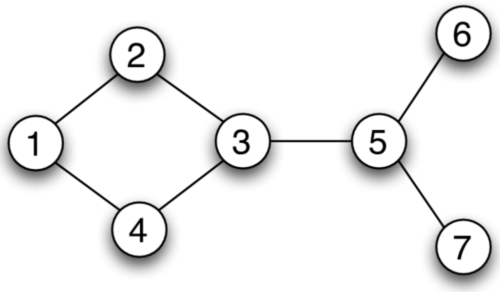

Figure 1: Graph

3.2 Types of Graph

3.2.1 Directed Graph

A directed graph, also known as a digraph, is a type of graph in which edges have a direction. It represents relationships between objects that have a cause-and-effect relationship, such as a food chain. For example, a food chain of lions, zebras, and grass can be represented as a directed graph where the edge points from zebras to lions, lions to grass, and grass to zebras.

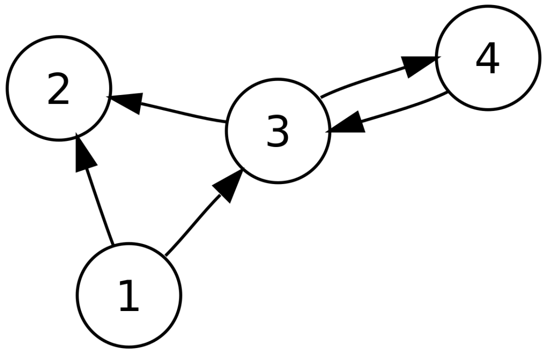

Figure 2: Directed Graph

3.2.2 Undirected Graph

An undirected graph is a type of graph in which edges have no direction. It represents relationships between objects that have a symmetric relationship, such as social networks. For example, a social network of friends can be represented as an undirected graph where each node represents a person, and an edge connects two people if they are friends.

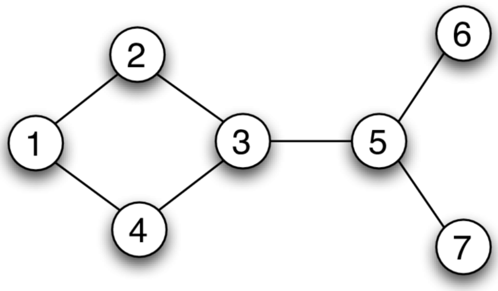

Figure 3: Undirected Graph

3.2.3 Weighted Graph

A weighted graph is a type of graph in which edges have a numerical value. It is used to represent relationships between objects where the relationship has a quantity associated with it, such as distances between cities. For example, a map of cities can be represented as a weighted graph where the nodes represent cities and the edges represent distances between them.

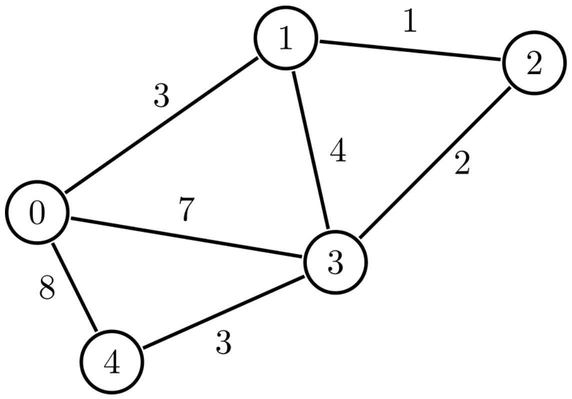

Figure 4: Weighted Graph

3.2.4 Bipartite Graph

A bipartite graph is a type of graph in which the nodes can be divided into two disjoint sets, such that each edge connects a node in one set to a node in the other set. It is used to represent relationships between two different sets of objects, such as employers and employees. For example, a company can be represented as a bipartite graph where one set of nodes represents employers and the other set represents employees, and an edge connects an employer to an employee if the employer employs the employee.

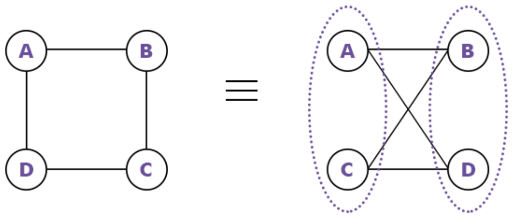

Figure 5: Bipartite Graph

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. Input at least 5 nodes.
2. Display DFT (recursive and non recursive) and BFT

6 Output

1. The traversal of the Threaded binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Creation of the Graph

```
//Pseudo Code for Creation of Threaded Binary Tree
```

8.2 Depth First Search (Recursive)

8.3 Depth First Search (Non Recursive)

8.4 Breadth First Search

9 Time Complexity

9.1 Creation of Threaded Binary Tree

- **Time Complexity:**
  $$O(V^2)$$
  for adjacency matrix representation
  where V is the number of vertices in the graph, and E is the number of edges in the graph.
- **Time Complexity:**
  $$O(E)$$
  for adjacency list representation
  where V is the number of vertices in the graph
- **Space Complexity:**
  $$O(V^2)$$
  for V being the number of vertices in the graph.

9.2 Recursive Depth First Traversal

- **Time Complexity:**
$$O(V+E)$$
for V being the number of vertices in the graph and E being the number of edges in the graph.
- **Space Complexity:**
$$O(V)$$
for V being the number of vertices in the graph.

9.3 Non Recursive Depth First Traversal

- **Time Complexity:**
$$O(V+E)$$
for V being the number of vertices in the graph and E being the number of edges in the graph.
- **Space Complexity:**
$$O(V)$$
for V being the number of vertices in the graph.

9.4 Breadth First Traversal

- **Time Complexity:**
$$O(V+E)$$
for V being the number of vertices in the graph and E being the number of edges in the graph.
- **Space Complexity:**
$$O(V)$$
for V being the number of vertices in the graph.

10 Code

10.1 Program

```cpp
// Creating of a Network in a Graph
#include <iostream>
#include <stack>

using namespace std;

// Class to store the vertex and its adjacent nodes
class GraphNode
{
    int vertex; // Stores the vertex
    GraphNode *next; // Pointer to the next node
    friend class Graph; // Making Graph class as a friend of GraphNode
};

// Graph class to create a graph and perform traversal operations
class Graph
```

```cpp
{
private:
    GraphNode *head[20]; // Array of pointers to store the head of each linked list
    int no_of_vtex; // Stores the number of vertices

public:
    // Constructor to initialize the number of vertices
    Graph(int no_of_vtex = 0)
    {
        this->no_of_vtex = no_of_vtex;
        for (int i = 0; i < no_of_vtex; i++)
        {
            head[i] = new GraphNode(); // Allocating memory for each head node
            head[i]->vertex = i; // Assigning the vertex value to the head node
        }
    }

    // Function to create the graph
    void create_graph()
    {
        int current_vertex;
        char choice;
        GraphNode *temp;
        for (int i = 0; i < no_of_vtex; i++)
        {
            temp = head[i];
            do
            {
                cout << "Enter the vertex to which " << i << " is connected" << endl;
                cin >> current_vertex;
                if (current_vertex == i)
                {
                    cout << "Self Loops are not allowed" << endl;
                }
                else
                {
                    GraphNode *current = new GraphNode(); // Allocating memory for the current node
                    current->vertex = current_vertex; // Assigning the vertex value to the current node
                    temp->next = current; // Linking the current node to the previous node
                    temp = temp->next; // Moving the pointer to the current node
                }
                cout << "Do you want to add more edges" << endl;
                cin >> choice;
            } while (choice == 'y' || choice == 'Y');
        }
    }

    // Function to perform Depth First Search using recursion
    void DFS_recursive()
    {
        int vertex;
        int visited[20];
        for (int i = 0; i < no_of_vtex; i++)
        {
            visited[i] = 0; // Initializing all the vertices as unvisited
        }
        cout << "What is the starting vertex" << endl;
        cin >> vertex;
        DFS_recursive_worker(vertex, visited); // Calling the recursive worker function
    }

    // Recursive worker function to traverse the graph
    void DFS_recursive_worker(int vertex, int visited[])
    {
        GraphNode *temp;
        temp = head[vertex];
        visited[vertex] = 1; // Marking the vertex as visited
        cout << vertex << " " << endl;
        for (int i = 0; i < no_of_vtex; i++)
        {
            if (visited[temp->vertex] == 0)
            {
                DFS_recursive_worker(temp->vertex, visited); // Recursively calling the worker function
            }
            temp = temp->next;
        }
    }

    // Function to perform Depth First Search without recursion
    void DFS_non_recursive(int vertex)
    {
        int visited[20];
        stack<int> s;
        for (int i = 0; i < no_of_vtex; i++)
            visited[i] = 0;
        s.push(vertex); // Pushing the starting vertex into the stack
        visited[vertex] = 1;
        do
        {
            vertex = s.top(); // Taking out the top element from the stack
            s.pop();
            cout << vertex << " ";
            for (int w = 0; w < no_of_vtex; w++)
            {
                if (!visited[w])
                {
                    s.push(w); // Pushing the unvisited vertex into the stack
                    visited[w] = 1;
                }
            }
        } while (!s.empty());
    }

    // Function to perform Breadth First Search
    void breadth_first_traversal()
    {
        int visited[20];
        int starting_vertex;
        for (int i = 0; i < no_of_vtex; i++)
        {
            visited[i] = 0; // Initializing all the vertices as unvisited
        }
        cout << "What is the starting vertex" << endl;
        cin >> starting_vertex;
        breadth_first_traversal(starting_vertex, visited); // Calling the BFS function
    }

    // Worker function to traverse the graph
    void breadth_first_traversal(int vertex, int visited[])
    {
        GraphNode *temp;
        temp = head[vertex];
        visited[vertex] = 1; // Marking the vertex as visited
        cout << vertex << " " << endl;
        for (int i = 0; i < no_of_vtex; i++)
        {
            if (visited[temp->vertex] == 0)
            {
                DFS_recursive_worker(temp->vertex, visited); // Calling the recursive worker function
            }
            temp = temp->next;
        }
    }
};

int main()
{
    int no_of_vtex, starting_vertex;
    cout << "Enter the number of vertices" << endl;
    cin >> no_of_vtex;
    Graph g(no_of_vtex);
    g.create_graph();
    cout << "Depth First Search Recursive" << endl;
    g.DFS_recursive();
    cout << "Depth First Search Non Recursive" << endl;
    cout << "What is the starting vertex" << endl;
    cin >> starting_vertex;
    g.DFS_non_recursive(starting_vertex);
    cout << "Breadth First Search" << endl;
    g.breadth_first_traversal();
    return 0;
}
```

10.2 Input and Output

```
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
```

```
8. Exit

1
Enter the Word: apple
Enter the definition of the word: fruit
Do you want to enter more Nodes? (0/1)

1
Enter the Word: banana
Enter the definition of the word: fruit
Do you want to enter another word? (1/0): 1
Enter the Word: keyboard
Enter the definition of the word: input
Do you want to enter another word? (1/0): 1
Enter the Word: pears
Enter the definition of the word: fruit
Do you want to enter another word? (1/0): 1
Enter the Word: bottle
Enter the definition of the word: water
Do you want to enter another word? (1/0): 1
Enter the Word: charger
Enter the definition of the word: charging
Do you want to enter another word? (1/0): 1
Enter the Word: monitor
Enter the definition of the word: see
Do you want to enter another word? (1/0): 1
Enter the Word: paper
Enter the definition of the word: 1
Do you want to enter another word? (1/0): 1
Enter the Word: pen
Enter the definition of the word: writing
Do you want to enter another word? (1/0): 1
Enter the Word: phone
Enter the definition of the word: scrolling
Do you want to enter another word? (1/0): 0

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

2
Traversing through the Binary Tree Inorder Iteratively:
apple : fruit
banana : fruit
bottle : water
charger : charging
keyboard : input
monitor : see
```

```
paper : 1
pears : fruit
pen : writing
phone : scrolling

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

5
Traversing through the Binary Tree using BFS:
apple : fruit
banana : fruit
keyboard : input
bottle : water
pears : fruit
charger : charging
monitor : see
pen : writing
paper : 1
phone : scrolling

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

6
Creating a copy of the tree
Traversing through the Binary Tree Inorder Iteratively:
apple : fruit
banana : fruit
bottle : water
charger : charging
keyboard : input
monitor : see
paper : 1
pears : fruit
```

```
pen : writing
phone : scrolling
Traversing via Breadth First Search:
apple : fruit
banana : fruit
keyboard : input
bottle : water
pears : fruit
charger : charging
monitor : see
pen : writing
paper : 1
phone : scrolling

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

7
Creating a mirror of the tree
Traversing through the Binary Tree Inorder Iteratively:
phone : scrolling
pen : writing
pears : fruit
paper : 1
monitor : see
keyboard : input
charger : charging
bottle : water
banana : fruit
apple : fruit
Traversing via Breadth First Search:
apple : fruit
banana : fruit
keyboard : input
pears : fruit
bottle : water
pen : writing
monitor : see
charger : charging
phone : scrolling
paper : 1

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

```

```
What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

8
Exiting the program
```

11 Conclusion

Thus, we have represented graph using adjacency list and performed DFT and BFT.

12 FAQ

1. **Explain two applications of graph.**

   Graphs have numerous applications in various fields such as computer science, social sciences, biology, transportation, and more. Here are two examples of applications of graphs:

   - **Social Networks**: Social networks such as Facebook, Twitter, and LinkedIn can be represented as graphs, where each user is a node, and the relationship between them (friendship, follow, connection, etc.) is an edge. Graph algorithms can be used to analyze and understand social networks, such as identifying clusters of friends, finding influential users, detecting fake accounts or spam, and predicting trends or user behavior. For example, graph analysis can help social networks to recommend new friends, suggest content to users, or optimize their algorithms to improve user engagement and retention.

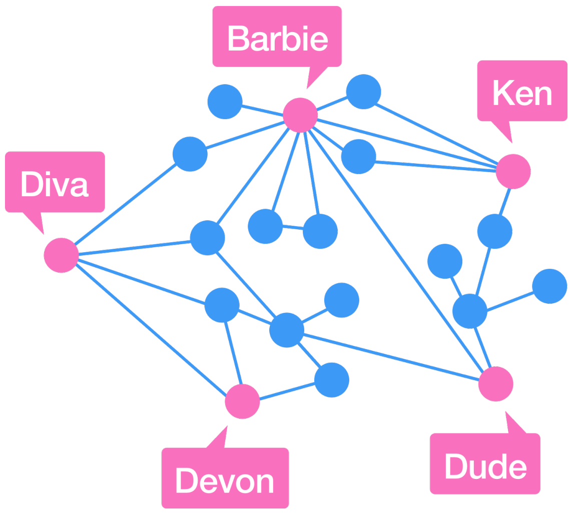

Figure 6: Graph as a Representation for Social Media

   - **Shortest Path Algorithms**: Graphs can be used to model transportation networks such as roads, railways, and flights, where each city or location is a node, and the roads or flights connecting them are edges. Shortest path algorithms such as Dijkstra's algorithm and the A* algorithm can be used to find the shortest path or the fastest route between two locations, taking into account factors such as distance, time, traffic, or cost. These algorithms are widely used in navigation systems, logistics, and transportation planning, and can help optimize the routing and scheduling of vehicles, goods, or passengers, leading to cost savings and improved efficiency.

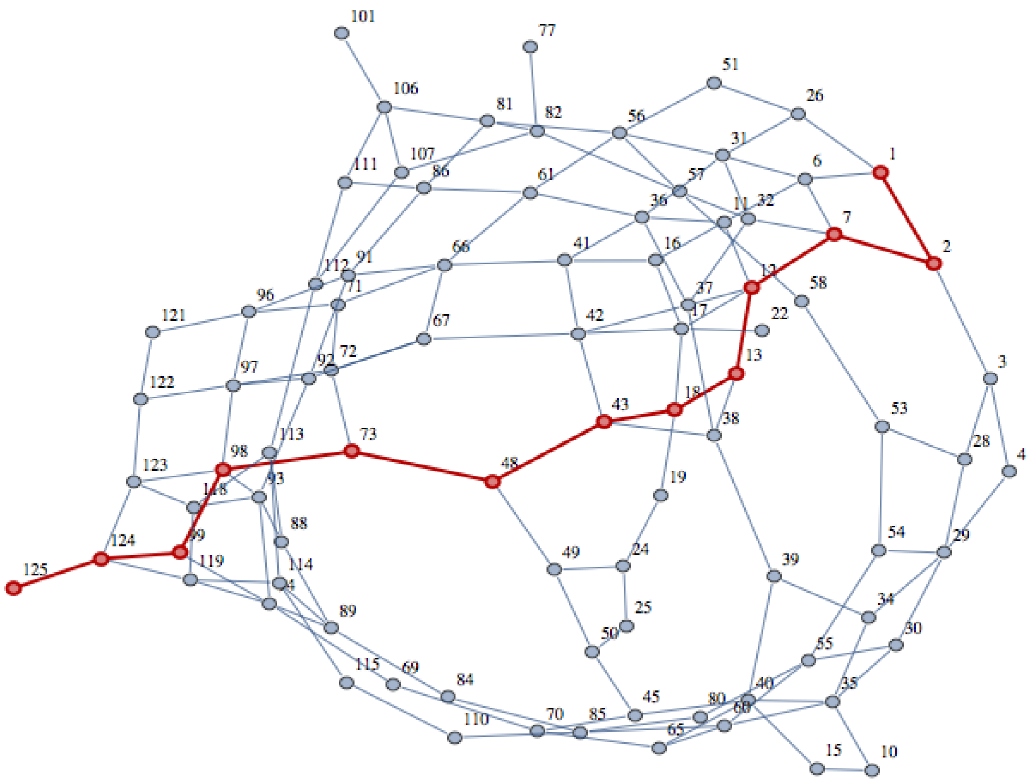

Figure 7: The Shortest Path Problem represented and solved using a Graph

2. **Explain advantages of adjacency list over adjacency matrix.**

   The adjacency list representation of a graph has the following advantages over an adjacency matrix representation:

   - **Space Efficient**: The adjacency list representation is more space efficient than the adjacency matrix representation. This is because the adjacency matrix representation of a graph with V vertices and E edges requires O(V^2) space, while the adjacency list representation requires O(V+E) space.
   - **Faster Adjacent Edge Lookups**: The adjacency list representation allows for faster lookups of adjacent edges of a vertex than the adjacency matrix representation. In the adjacency matrix representation, to find the adjacent edges of a vertex, we have to scan through the entire row of the vertex. In the adjacency list representation, we can simply traverse the linked list of the vertex to find its adjacent edges.
   - **Faster Degree Lookups**: The adjacency list representation allows for faster lookups of the degree of a vertex than the adjacency matrix representation. In the adjacency matrix representation, to find the degree of a vertex, we have to scan through the entire row of the vertex. In the adjacency list representation, we can simply traverse the linked list of the vertex to find its degree.

3. **Why transversal in graph is different than traversal in tree**

- **Graphs can have cycles**: Unlike trees, graphs can contain cycles, which means that a node can be visited multiple times through different paths. This makes traversal more complex, as we need to keep track of the visited nodes to avoid infinite loops or redundant computations. In trees, on the other hand, there are no cycles, so each node is visited exactly once during traversal.
- **Graphs can be disconnected**: Graphs can have disconnected components, which means that some nodes may not be reachable from others. This means that traversal of a graph needs to handle the possibility of having multiple disconnected components, and ensure that all components are visited. In contrast, trees are always connected, so there is no need to worry about disconnected components.
- **Graphs can be directed or undirected**: Graphs can be either directed (where edges have a direction) or undirected (where edges have no direction), which affects how traversal algorithms operate. For example, in a directed graph, traversal algorithms need to take into account the direction of edges to avoid going back to already-visited nodes, while in an undirected graph, traversal can simply use a marking mechanism to track visited nodes.