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

April 16, 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.*

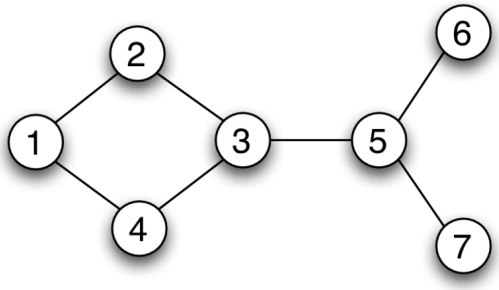


Figure 1: Graph

## 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.*

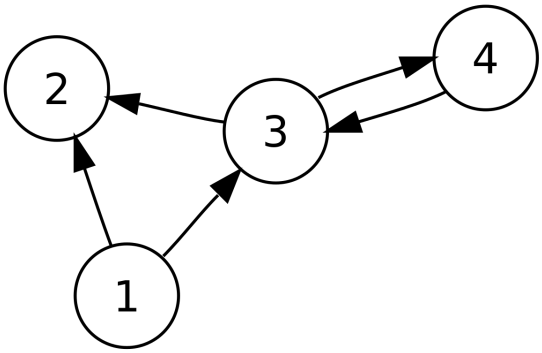


Figure 2: Directed Graph

### 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.*

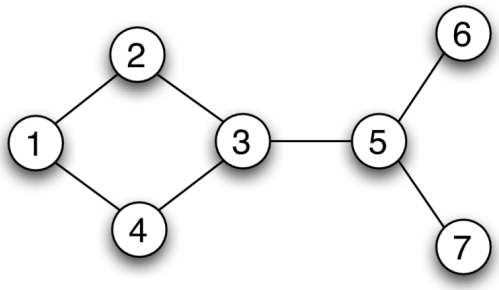


Figure 3: Undirected Graph

### 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.*

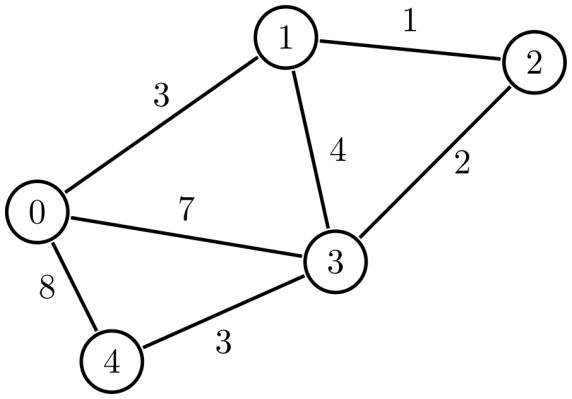


Figure 4: Weighted Graph

### 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.*

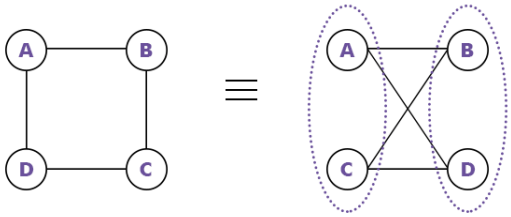


Figure 5: Bipartite Graph

## 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

```
1
2  void create_graph()
3  {
4      int src, dst;
5      char ch;
6      do
7          cout << "Enter the source and destination" << endl;
8          cin >> src >> dst;
9          if (src >= nodes || dst >= nodes || src < 0 || dst < 0)
10             cout << "Invalid edge" << endl;
11             continue;
12         if (src == dst)
13             cout << "Invalid edge, self loops not allowed." << endl;
14             continue;
15         AddEdge(src, dst);
16         cout << "Do you want to continue" << endl;
17         cin >> ch;
18     } while (ch == 'y' || ch == 'Y');
19 }
```

## 8.2 Depth First Search (Recursive)

```
1  void DFS_recursive()
2  {
3      bool *visited = new bool[nodes];
4      for (int i = 0; i < nodes; i++)
5          visited[i] = false;
6      for (int i = 0; i < nodes; i++)
7          if (visited[i] == false)
8              DFS_recursive(i, visited);
9      cout << endl;
10 }
11
12 void DFS_recursive(int src, bool *visited)
13 {
14     visited[src] = true;
15     cout << src << " ";
16     for (auto &adj_node : adjlist[src])
17         if (visited[adj_node] == false)
18             DFS_recursive(adj_node, visited);
19 }
```

### 8.3 Breadth First Search

```
void breadth_first_traversal()
{
    bool *visited = new bool[nodes];
    for (int i = 0; i < nodes; i++)
        visited[i] = false;
    for (int i = 0; i < nodes; i++)
        if (visited[i] == false)
            breadth_first_traversal(i, visited);
    cout << endl;
}
```

## 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

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

using namespace std;

class Graph
{

private:
    int nodes;
    list<int> *adjlist;

public:
    Graph()
    {
    }

    Graph(int nodes)
    { // Allocate resources
        adjlist = new list<int>[nodes];
        this->nodes = nodes;
    }

    ~Graph()
    { // Free allocated resources
        delete[] adjlist;
    }

    void AddEdge(int src, int dst)
    {
        adjlist[src].push_back(dst);
        adjlist[dst].push_back(src);
    }

    void Iterate(int src)
    {
        cout << src << " : ";
        for (auto &adj_node : adjlist[src])
```

```
42          {
43              cout << adj_node << " ";
44          }
45          cout << endl;
46      }
47
48      void DFS_recursive()
49      {
50          bool *visited = new bool[nodes];
51          for (int i = 0; i < nodes; i++)
52          {
53              visited[i] = false;
54          }
55          for (int i = 0; i < nodes; i++)
56          {
57              if (visited[i] == false)
58              {
59                  DFS_recursive(i, visited);
60              }
61          }
62          cout << endl;
63      }
64
65      void DFS_recursive(int src, bool *visited)
66      {
67          visited[src] = true;
68          cout << src << " ";
69          for (auto &adj_node : adjlist[src])
70          {
71              if (visited[adj_node] == false)
72              {
73                  DFS_recursive(adj_node, visited);
74              }
75          }
76      }
77
78      void breadth_first_traversal()
79      {
80          bool *visited = new bool[nodes];
81          for (int i = 0; i < nodes; i++)
82          {
83              visited[i] = false;
84          }
85          for (int i = 0; i < nodes; i++)
86          {
87              if (visited[i] == false)
88              {
89                  breadth_first_traversal(i, visited);
90              }
91          }
92          cout << endl;
93      }
94
95      void breadth_first_traversal(int src, bool *visited)
96      {
97          queue<int> q;
98          q.push(src);
99          visited[src] = true;
100         while (!q.empty())
```

```
        {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (auto &adj_node : adjlist[node])
            {
                if (visited[adj_node] == false)
                {
                    q.push(adj_node);
                    visited[adj_node] = true;
                }
            }
        }
    }

    void create_graph()
    {
        int src, dst;
        char ch;
        do
        {
            cout << "Enter the source and destination" << endl;
            cin >> src >> dst;
            if (src >= nodes || dst >= nodes || src < 0 || dst < 0)
            {
                cout << "Invalid edge" << endl;
                continue;
            }
            if (src == dst)
            {
                cout << "Invalid edge, self loops not allowed." << endl;
                continue;
            }
            AddEdge(src, dst);
            cout << "Do you want to continue" << endl;
            cin >> ch;
        } while (ch == 'y' || ch == 'Y');
    }
};

int main()
{
    Graph g(10);

    // g.AddEdge(0, 1);
    // g.AddEdge(0, 2);
    // g.AddEdge(1, 3);
    // g.AddEdge(1, 4);
    // g.AddEdge(2, 3);
    // g.AddEdge(3, 5);
    // g.AddEdge(4, 6);
    // g.AddEdge(5, 6);
    // g.AddEdge(5, 7);
    // g.AddEdge(6, 7);
    // g.AddEdge(6, 8);
    // g.AddEdge(7, 8);
    // g.AddEdge(7, 9);
    // g.AddEdge(8, 9);

```

```
    // cout << "Adjacency list implementation for graph" << endl;

    // g.Iterate(0);
    // g.Iterate(1);
    // g.Iterate(4);

    g.create_graph();

    cout << "Depth First Search Recursive" << endl;
    g.DFS_recursive();
    cout << "Breadth First Search" << endl;
    g.breadth_first_traversal();

    return 0;
}
```

## 10.2 Input and Output

```
Enter the source and destination
0 1
Do you want to continue
y
Enter the source and destination
0 2
Do you want to continue
y
Enter the source and destination
1 3
Do you want to continue
y
Enter the source and destination
1 4
Do you want to continue
y
Enter the source and destination
2 3
Do you want to continue
y
Enter the source and destination
3 5
Do you want to continue
y
Enter the source and destination
4 6
Do you want to continue
y
Enter the source and destination
5 6
Do you want to continue
y
Enter the source and destination
5 7
Do you want to continue
y
Enter the source and destination
6 7
Do you want to continue
y
Enter the source and destination
```

```
6 8
Do you want to continue
y
Enter the source and destination
7 8
Do you want to continue
y
Enter the source and destination
7 9
Do you want to continue
y
Enter the source and destination
8 9
Do you want to continue
n
Depth First Search Recursive
0 1 3 2 5 6 4 7 8 9
Breadth First Search
0 1 2 3 4 5 6 7 8 9
```

## 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.

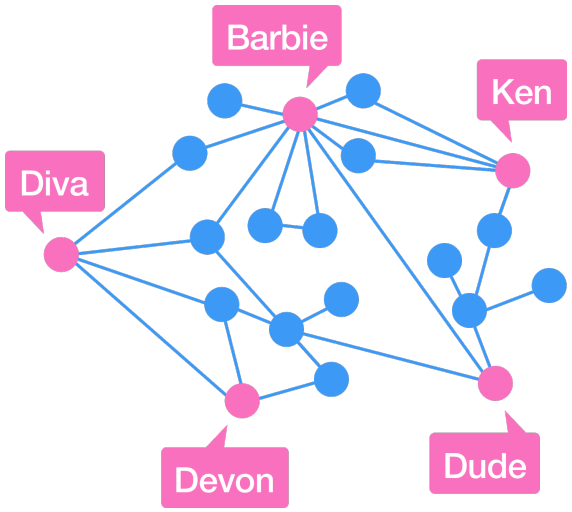


Figure 6: Graph as a Representation for Social Media

   - **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.

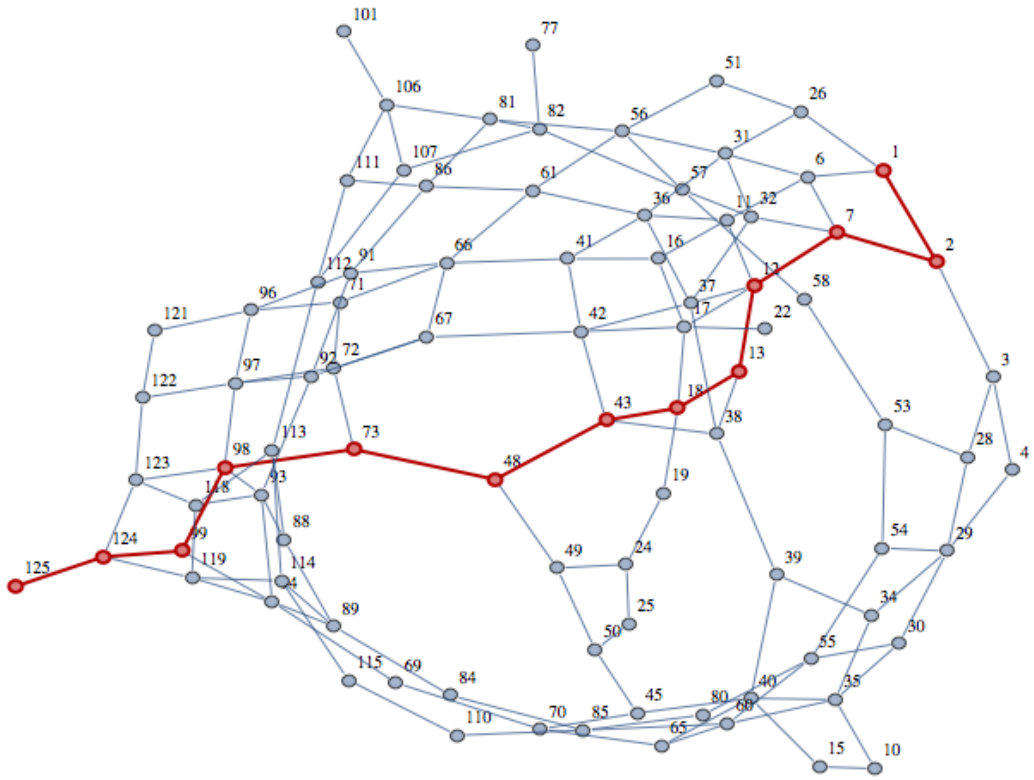


Figure 7: The Shortest Path Problem represented and solved using a Graph

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.