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 19, 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.
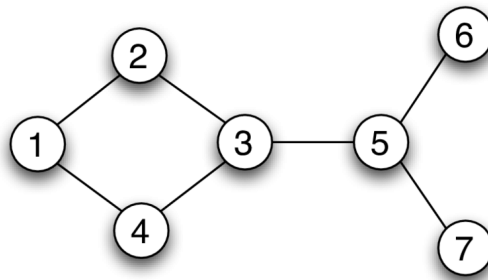
Figure 1: Graph

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.
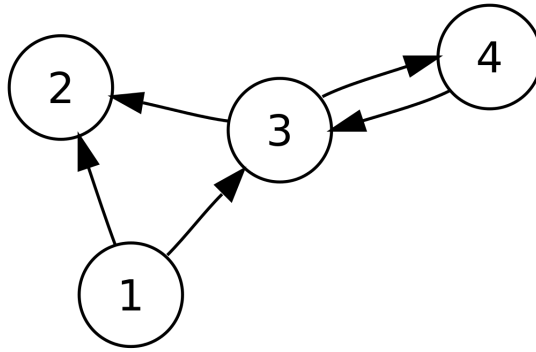
Figure 2: Directed Graph

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.
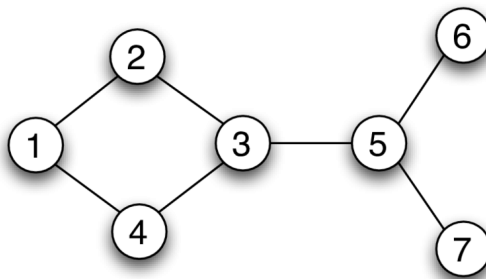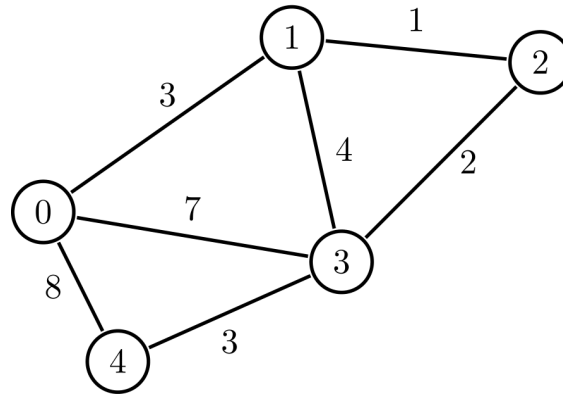
Figure 4: Weighted Graph

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.
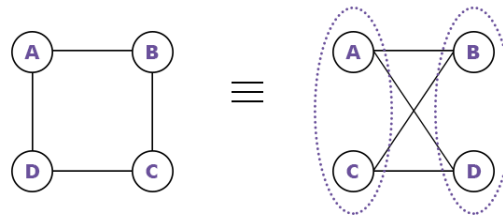
Figure 5: Bipartite Graph

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    //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 GraphNode
{
    int vertex;
    GraphNode *next;
    friend class Graph;
};

class Graph
{
private:
```

```cpp
17      GraphNode *head[20];
18      int no_of_vtex;
19      char choice;
20
21  public:
22      Graph()
23      {
24          for (int i = 0; i < no_of_vtex; i++)
25          {
26              head[i] = new GraphNode;
27              head[i]->vertex = i;
28          }
29      }
30      void create_graph()
31      {
32          int current_vertex;
33          int choice;
34          GraphNode *temp;
35          for (int i = 0; i < no_of_vtex; i++)
36          {
37              temp = head[i];
38              do
39              {
40                  cin >> current_vertex;
41                  if (current_vertex == i)
42                  {
43                      cout << "Self Loops are not allowed" << endl;
44                  }
45                  else
46                  {
47                      GraphNode *current = new GraphNode;
48                      current->vertex = current_vertex;
49                      temp->next = current;
50                      temp = temp->next;
51                  }
52                  cin >> choice;
53              } while (choice == 'y' || choice == 'Y');
54          }
55      }
56
57      void depth_first_search(int vertex)
58      {
59          int visited[20];
60          for (int i = 0; i < no_of_vtex; i++)
61          {
62              visited[i] = 0;
63          }
64          cout << "What is the starting vertex" << endl;
65          cin >> vertex;
66          dfs(vertex, visited);
67      }
68
69      void dfs(int vertex, int visited[])
70      {
71          GraphNode *temp;
72          temp = head[vertex];
73          visited[vertex] = 1;
74          cout << vertex << " " << endl;
75          for (int i = 0; i < no_of_vtex; i++)
```

```
76            {
77                if (visited[temp->vertex] == 0)
78                {
79                    dfs(temp->vertex, visited);
80                }
81                temp = temp->next;
82            }
83        }
84
85        void DFS_non_recursive(int vertex)
86        {
87            int visited[20];
88            stack<int> s;
89            for (int i = 0; i < no_of_vtex; i++)
90            {
91                visited[i] = 0;
92            }
93            s.push(vertex);
94            visited[vertex] = 1;
95            do
96            {
97                vertex = s.top();
98                s.pop();
99                cout << vertex << " ";
100               for (int w = 0; w < no_of_vtex; w++)
101               {
102                   if (!visited[w])
103                   {
104                       s.push(w);
105                       visited[w] = 1;
106                   }
107               }
108           } while (!s.empty());
109       }
110
111       void breadth_first_traversal(){
112           int visited[20];
113           for (int i = 0; i < no_of_vtex; i++)
114           {
115               visited[i] = 0;
116           }
117           cout << "What is the starting vertex" << endl;
118           cin >> vertex;
119           bfs(vertex, visited);
120       }
121 };
122
123 int main()
124 {
125     Graph g;
126     g.create_graph();
127     g.depth_first_search(0);
128     g.DFS_non_recursive(0);
129     return 0;
130 }
```

10.2   Input and Output

```
1 What would like to do?
```

```
 2
 3
 4  Welcome to ADS Assignment 2 - Binary Tree Traversals
 5
 6  What would you like to do?
 7  1. Create a Binary Search Tree
 8  2. Traverse the Tree Inorder Iteratively
 9  3. Traverse the Tree PreOrder Iteratively
10  4. Traverse the Tree PostOrder Iteratively
11  5. Traverse it using BFS
12  6. Create a Copy of the tree Recursively
13  7. Create a Mirror of the Tree Recursively
14  8. Exit
15
16  1
17  Enter the Word: apple
18  Enter the definition of the word: fruit
19  Do you want to enter more Nodes? (0/1)
20
21  1
22  Enter the Word: banana
23  Enter the definition of the word: fruit
24  Do you want to enter another word? (1/0): 1
25  Enter the Word: keyboard
26  Enter the definition of the word: input
27  Do you want to enter another word? (1/0): 1
28  Enter the Word: pears
29  Enter the definition of the word: fruit
30  Do you want to enter another word? (1/0): 1
31  Enter the Word: bottle
32  Enter the definition of the word: water
33  Do you want to enter another word? (1/0): 1
34  Enter the Word: charger
35  Enter the definition of the word: charging
36  Do you want to enter another word? (1/0): 1
37  Enter the Word: monitor
38  Enter the definition of the word: see
39  Do you want to enter another word? (1/0): 1
40  Enter the Word: paper
41  Enter the definition of the word: 1
42  Do you want to enter another word? (1/0): 1
43  Enter the Word: pen
44  Enter the definition of the word: writing
45  Do you want to enter another word? (1/0): 1
46  Enter the Word: phone
47  Enter the definition of the word: scrolling
48  Do you want to enter another word? (1/0): 0
49
50  What would like to do?
51
52
53  Welcome to ADS Assignment 2 - Binary Tree Traversals
54
55  What would you like to do?
56  1. Create a Binary Search Tree
57  2. Traverse the Tree Inorder Iteratively
58  3. Traverse the Tree PreOrder Iteratively
59  4. Traverse the Tree PostOrder Iteratively
60  5. Traverse it using BFS
```

```
61  6. Create a Copy of the tree Recursively
62  7. Create a Mirror of the Tree Recursively
63  8. Exit
64
65  2
66  Traversing through the Binary Tree Inorder Iteratively:
67  apple : fruit
68  banana : fruit
69  bottle : water
70  charger : charging
71  keyboard : input
72  monitor : see
73  paper : 1
74  pears : fruit
75  pen : writing
76  phone : scrolling
77
78  What would like to do?
79
80
81  Welcome to ADS Assignment 2 - Binary Tree Traversals
82
83  What would you like to do?
84  1. Create a Binary Search Tree
85  2. Traverse the Tree Inorder Iteratively
86  3. Traverse the Tree PreOrder Iteratively
87  4. Traverse the Tree PostOrder Iteratively
88  5. Traverse it using BFS
89  6. Create a Copy of the tree Recursively
90  7. Create a Mirror of the Tree Recursively
91  8. Exit
92
93  5
94  Traversing through the Binary Tree using BFS:
95  apple : fruit
96  banana : fruit
97  keyboard : input
98  bottle : water
99  pears : fruit
100  charger : charging
101  monitor : see
102  pen : writing
103  paper : 1
104  phone : scrolling
105
106  What would like to do?
107
108
109  Welcome to ADS Assignment 2 - Binary Tree Traversals
110
111  What would you like to do?
112  1. Create a Binary Search Tree
113  2. Traverse the Tree Inorder Iteratively
114  3. Traverse the Tree PreOrder Iteratively
115  4. Traverse the Tree PostOrder Iteratively
116  5. Traverse it using BFS
117  6. Create a Copy of the tree Recursively
118  7. Create a Mirror of the Tree Recursively
119  8. Exit
```

```
120
121 6
122 Creating a copy of the tree
123 Traversing through the Binary Tree Inorder Iteratively:
124 apple : fruit
125 banana : fruit
126 bottle : water
127 charger : charging
128 keyboard : input
129 monitor : see
130 paper : 1
131 pears : fruit
132 pen : writing
133 phone : scrolling
134 Traversing via Breadth First Search:
135 apple : fruit
136 banana : fruit
137 keyboard : input
138 bottle : water
139 pears : fruit
140 charger : charging
141 monitor : see
142 pen : writing
143 paper : 1
144 phone : scrolling
145
146 What would like to do?
147
148
149 Welcome to ADS Assignment 2 - Binary Tree Traversals
150
151 What would you like to do?
152 1. Create a Binary Search Tree
153 2. Traverse the Tree Inorder Iteratively
154 3. Traverse the Tree PreOrder Iteratively
155 4. Traverse the Tree PostOrder Iteratively
156 5. Traverse it using BFS
157 6. Create a Copy of the tree Recursively
158 7. Create a Mirror of the Tree Recursively
159 8. Exit
160
161 7
162 Creating a mirror of the tree
163 Traversing through the Binary Tree Inorder Iteratively:
164 phone : scrolling
165 pen : writing
166 pears : fruit
167 paper : 1
168 monitor : see
169 keyboard : input
170 charger : charging
171 bottle : water
172 banana : fruit
173 apple : fruit
174 Traversing via Breadth First Search:
175 apple : fruit
176 banana : fruit
177 keyboard : input
178 pears : fruit
```

```
179 bottle : water
180 pen : writing
181 monitor : see
182 charger : charging
183 phone : scrolling
184 paper : 1
185
186 What would like to do?
187
188
189 Welcome to ADS Assignment 2 - Binary Tree Traversals
190
191 What would you like to do?
192 1. Create a Binary Search Tree
193 2. Traverse the Tree Inorder Iteratively
194 3. Traverse the Tree PreOrder Iteratively
195 4. Traverse the Tree PostOrder Iteratively
196 5. Traverse it using BFS
197 6. Create a Copy of the tree Recursively
198 7. Create a Mirror of the Tree Recursively
199 8. Exit
200
201 8
202 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.
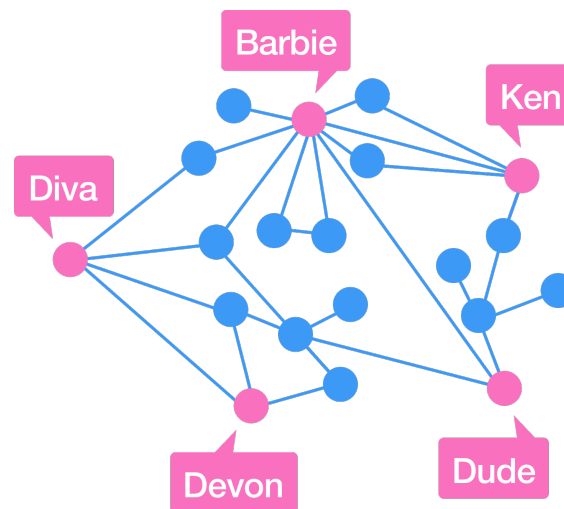

Figure 6: Graph as a Representation for Social Media

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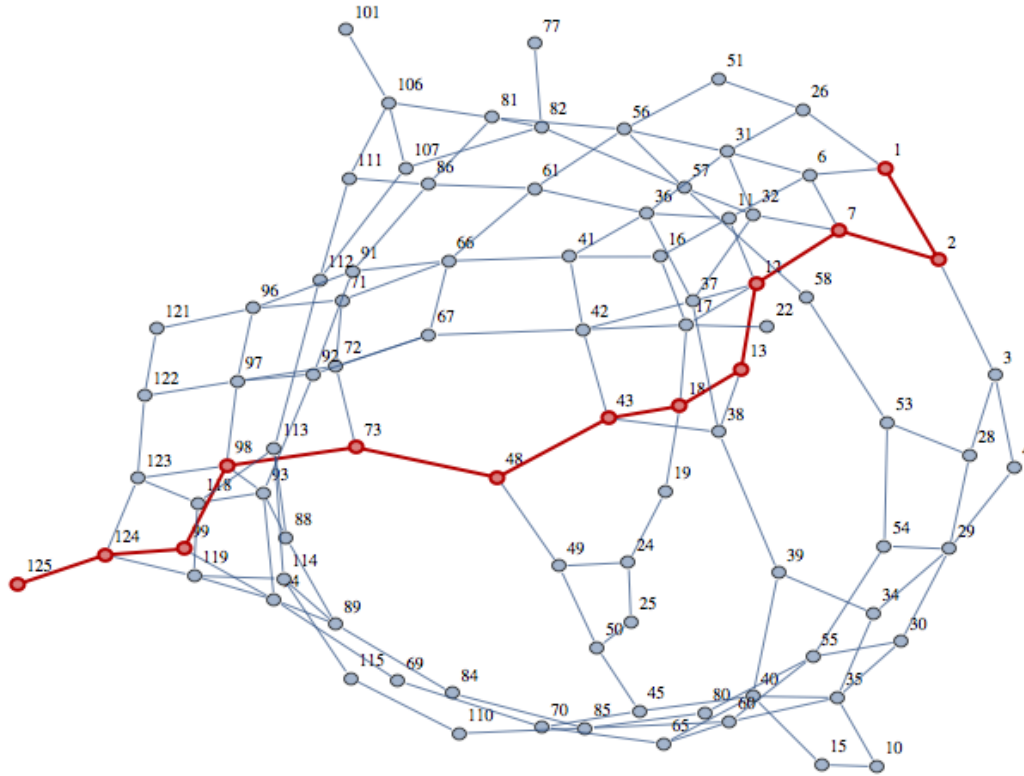
Figure 7: The Shortest Path Problem represented and solved using a Graph

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.