DATA STRUCTURES AND ALGORITHMS LABORATORY

GROUP C
ASSIGNMENT NO. 1

Name   :- Ojus Pravin Jaiswal

Roll No. :- SACO19108

Division :- A

Group C

Assignment 1

Title: Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS.

Objectives:

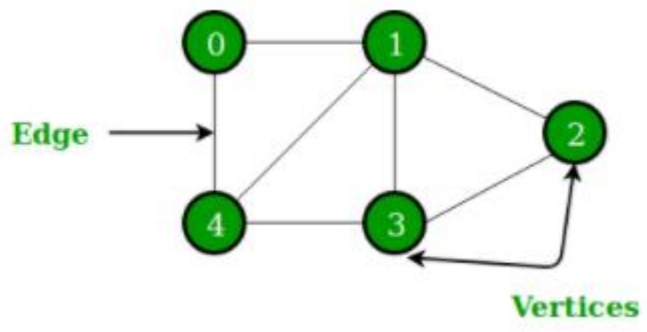
1. To identify directed and undirected graph.
2. To represent graph using adjacency matrix and list.
3. To traverse the graph.

Outcome:

1. To implement program to represent graph using adjacency matrix and list.
2. To implement program to traverse the graph.

Theory:

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.



A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

The following two are the most commonly used representations of a graph.

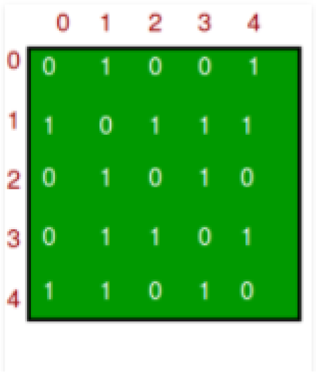
1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

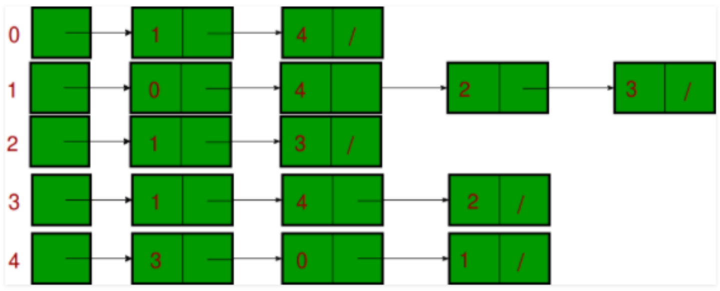
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



Breadth First Search or BFS is a graph traversal algorithm.

- It is used for traversing or searching a graph in a systematic fashion.
- BFS uses a strategy that searches in the graph in breadth first manner whenever possible.

- Queue data structure is used in the implementation of breadth first search.

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. raphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

Algorithm:

- Create a recursive function that takes the index of node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

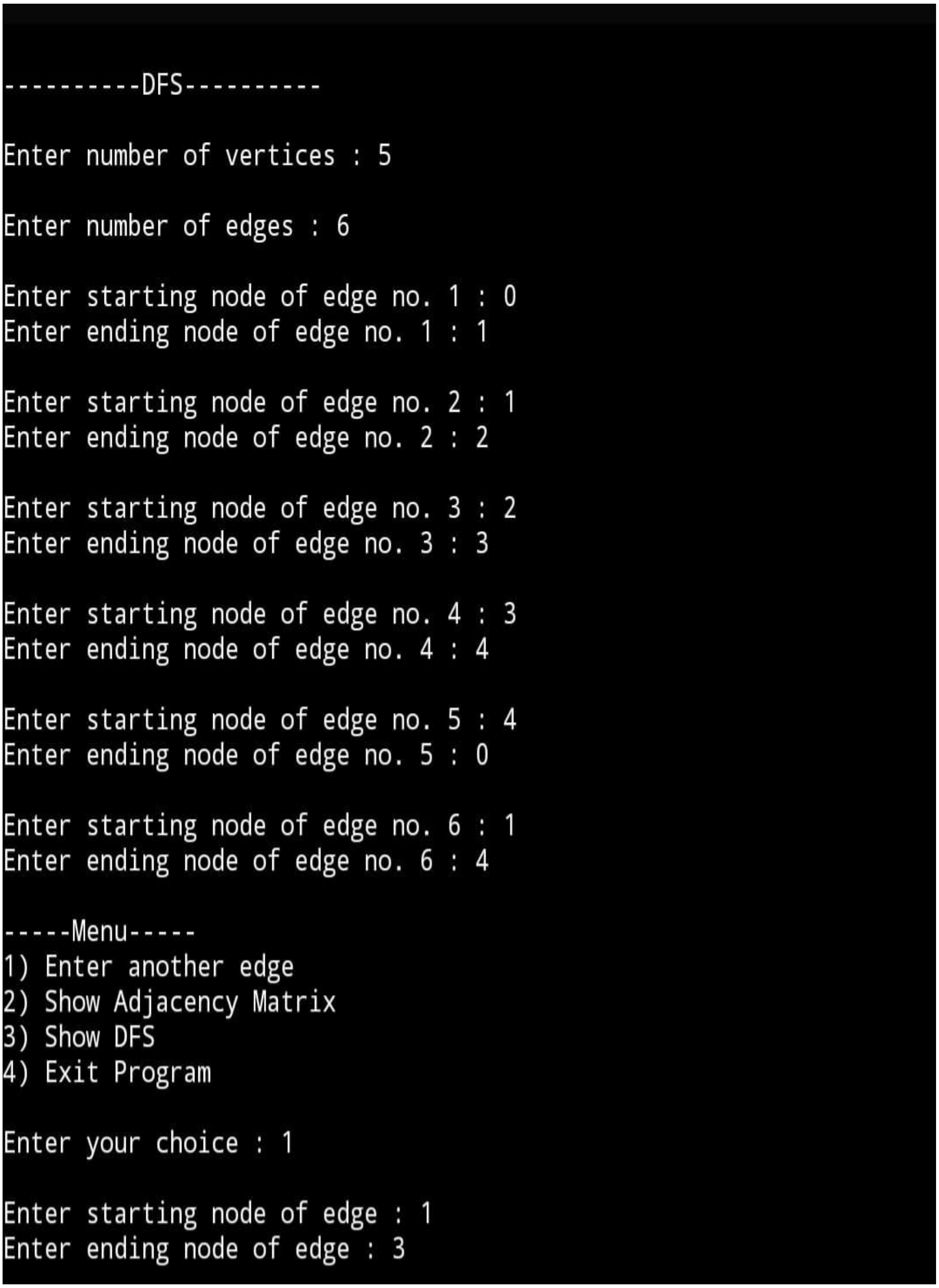
Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input: Vertices & Edges

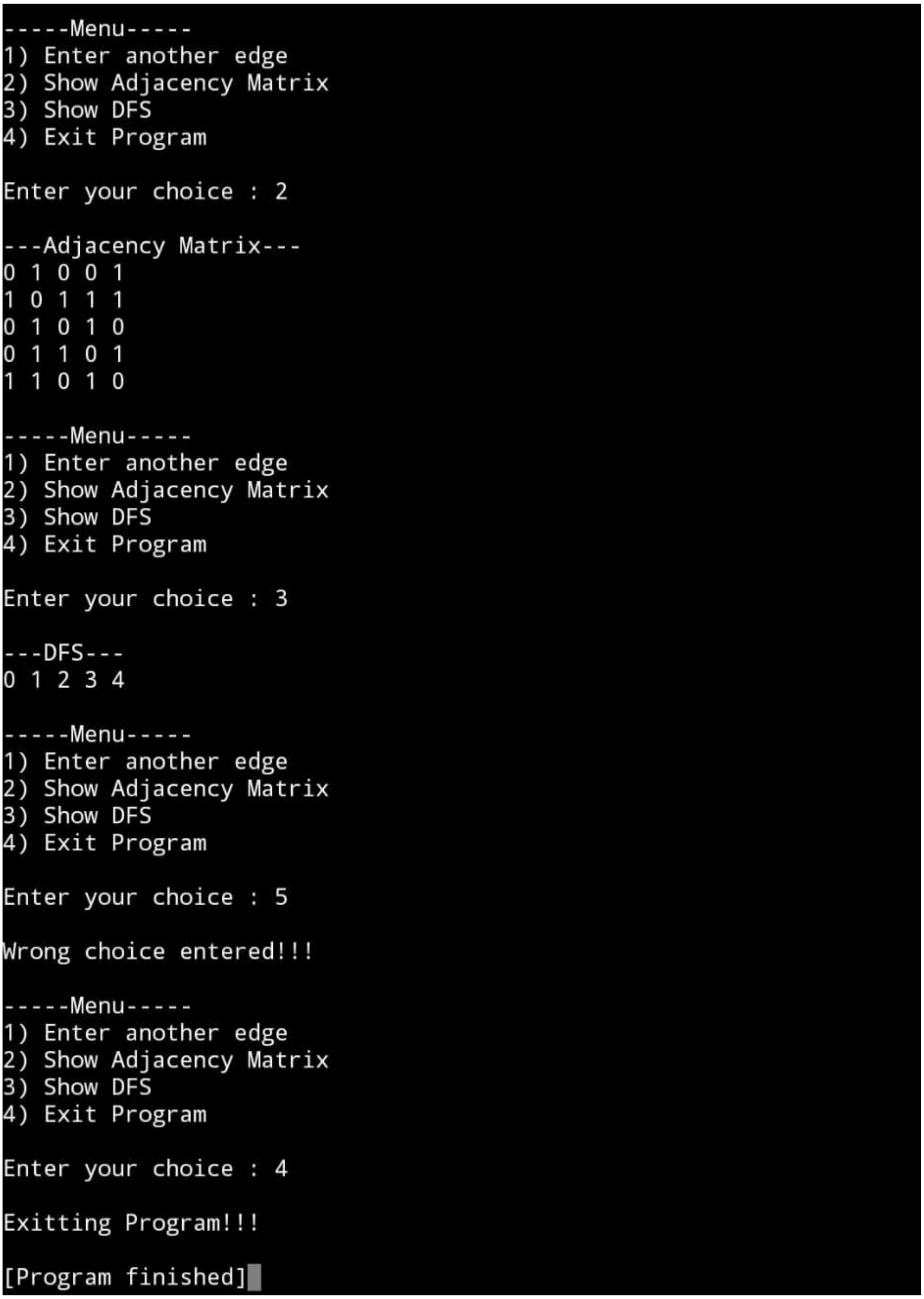
Program 1:

```
#include <iostream>
#include <vector>
using namespace std;

class Graph
{
        // Number of vertex
        int v;

        // Number of edges
        int e;

        // Adjacency matrix
        int **adj;

  public:
        // To create the initial adjacency matrix
        Graph(int v, int e);

        // Function to insert a new edge
        void addEdge(int start, int e);

        void print();

        // Function to display the DFS traversal
        void DFS(int start, vector<bool> &visited);
};

// Function to fill the empty adjacency matrix
Graph::Graph(int v, int e)
{
        this->v = v;
        this->e = e;
        adj = new int *[v];
        for (int row = 0; row < v; row++)
        {
                adj[row] = new int[v];
                for (int column = 0; column < v; column++)
                {
                        adj[row][column] = 0;
                }
        }
}

// Function to add an edge to the graph
void Graph::addEdge(int start, int e)
{
        // Considering a bidirectional edge
        adj[start][e] = 1;
        adj[e][start] = 1;
}

void Graph::print()
{
        // printing the matrix
        for (int i = 0; i < v; i++)
        {
                for (int j = 0; j < v; j++)
                {
                        cout << adj[i][j] << " ";
                }
                cout << endl;
        }
}

// Function to perform DFS on the graph
void Graph::DFS(int start, vector<bool> &visited)
{
        // Print the current node
        cout << start << " ";

        // Set current node as visited
        visited[start] = true;

        // For every node of the graph
        for (int i = 0; i < v; i++)
        {
                // If some node is adjacent to the current node
                // and it has not already been visited
                if (adj[start][i] == 1 && (!visited[i]))
                {
                        DFS(i, visited);
                }
        }
}

// Driver code
int main()
{
        int v, e, start, end, c;
        cout << "\n----------DFS----------\n";
        cout << "\nEnter number of vertices : ";
        cin >> v;
        cout << "\nEnter number of edges : ";
        cin >> e;
        // Create the graph
        Graph G(v, e);
        for (int i = 0; i < e; i++)
        {
                cout << "\nEnter starting node of edge no. " << i + 1 << " : ";
                cin >> start;
                cout << "Enter ending node of edge no. " << i + 1 << " : ";
                cin >> end;
                G.addEdge(start, end);
        }
        while (1)
        {
                // Visited vector to so that
                // a vertex is not visited more than once
                // Initializing the vector to false as no
                // vertex is visited at the beginning
                vector<bool> visited(v, false);
                cout << "\n-----Menu-----\n1) Enter another edge\n2) Show Adjacency Matrix\n3) Show
DFS\n4) Exit Program\n";
                cout << "\nEnter your choice : ";
                cin >> c;
                switch (c)
                {
                case 1:
                        cout << "\nEnter starting node of edge : ";
                        cin >> start;
                        cout << "Enter ending node of edge : ";
                        cin >> end;
                        G.addEdge(start, end);
                        break;
                case 2:
                        cout << "\n---Adjacency Matrix---\n";
                        G.print();
                        break;
                case 3:
                        // Perform DFS
                        cout << "\n---DFS---\n";
                        G.DFS(0, visited);
                        cout << endl;
                        break;
                case 4:
                        cout << "\nExitting Program!!!\n";
                        exit(0);
                default:
                        cout << "\nWrong choice entered!!!\n";
                }
        }
        return (0);
}
```

Output 1:



```
----------DFS----------

Enter number of vertices : 5

Enter number of edges : 6

Enter starting node of edge no. 1 : 0
Enter ending node of edge no. 1 : 1

Enter starting node of edge no. 2 : 1
Enter ending node of edge no. 2 : 2

Enter starting node of edge no. 3 : 2
Enter ending node of edge no. 3 : 3

Enter starting node of edge no. 4 : 3
Enter ending node of edge no. 4 : 4

Enter starting node of edge no. 5 : 4
Enter ending node of edge no. 5 : 0

Enter starting node of edge no. 6 : 1
Enter ending node of edge no. 6 : 4

-----Menu-----
1) Enter another edge
2) Show Adjacency Matrix
3) Show DFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge : 1
Enter ending node of edge : 3
```



```
-----Menu-----
1) Enter another edge
2) Show Adjacency Matrix
3) Show DFS
4) Exit Program

Enter your choice : 2

---Adjacency Matrix---
0 1 0 0 1
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
1 1 0 1 0

-----Menu-----
1) Enter another edge
2) Show Adjacency Matrix
3) Show DFS
4) Exit Program

Enter your choice : 3

---DFS---
0 1 2 3 4

-----Menu-----
1) Enter another edge
2) Show Adjacency Matrix
3) Show DFS
4) Exit Program

Enter your choice : 5

Wrong choice entered!!!

-----Menu-----
1) Enter another edge
2) Show Adjacency Matrix
3) Show DFS
4) Exit Program

Enter your choice : 4

Exitting Program!!!

[Program finished]
```

Program 2:

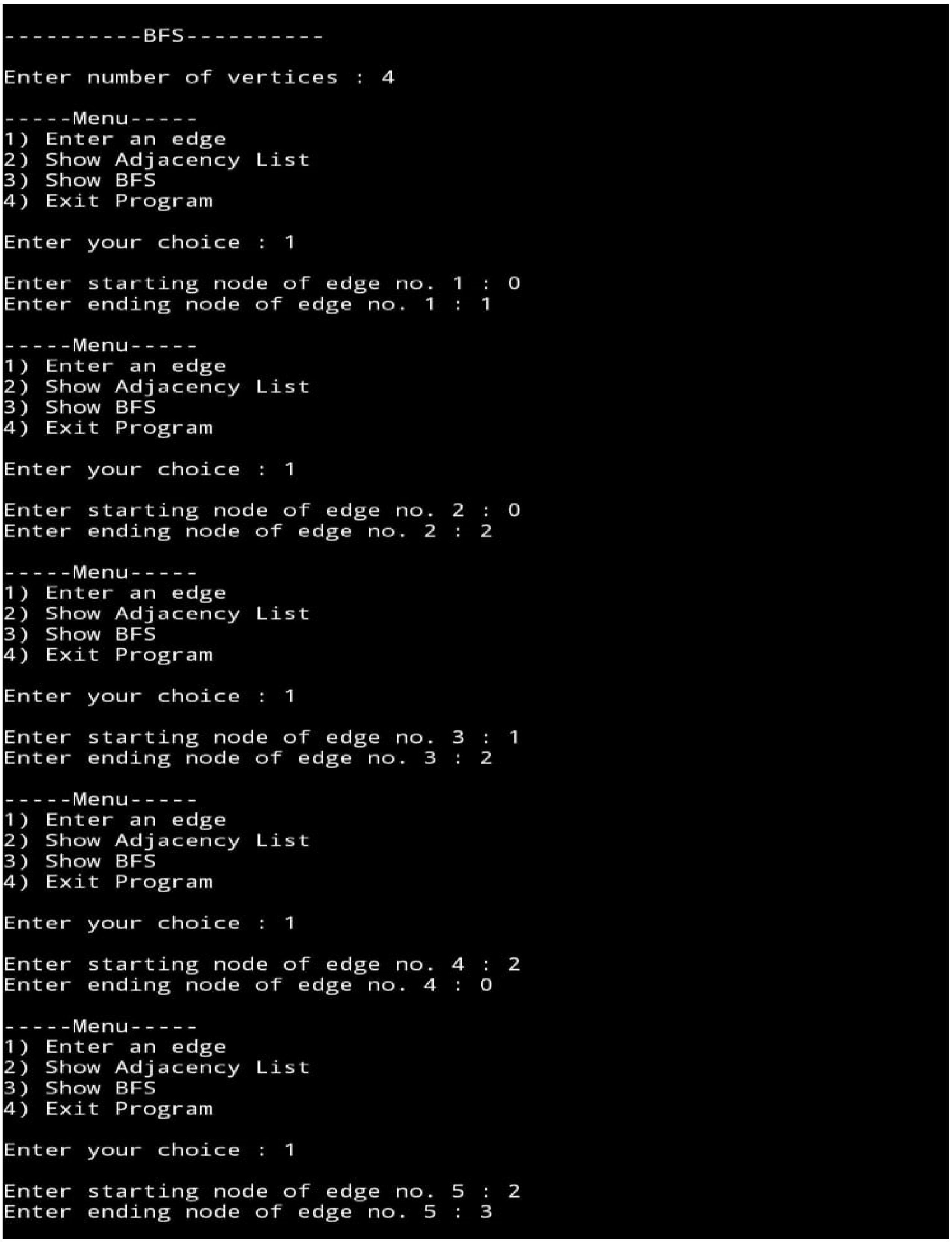
```
#include <iostream>
#include <list>
using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
        int V; // No. of vertices

        // Pointer to an array containing adjacency
        // lists
        list<int> *adj;

  public:
        Graph(int V); // Constructor

        // function to add an edge to graph
        void addEdge(int v, int w);

        // function to print
        void print();
        // prints BFS traversal from a given source s
        void BFS(int s);
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
```

```
{
        adj[v].push_back(w); // Add w to v’s list.
}
void Graph::print()
{
        list<int>::iterator i;
        for (int j = 0; j < V; j++)
        {
                cout << "adj[" << j << "]";
                for (i = adj[j].begin(); i != adj[j].end(); ++i)
                {
                        cout << "-->" << *i;
                }
                cout << endl;
        }
}
void Graph::BFS(int s)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;

        // Create a queue for BFS
        list<int> queue;

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.push_back(s);

        // 'i' will be used to get all adjacent
        // vertices of a vertex
        list<int>::iterator i;
```

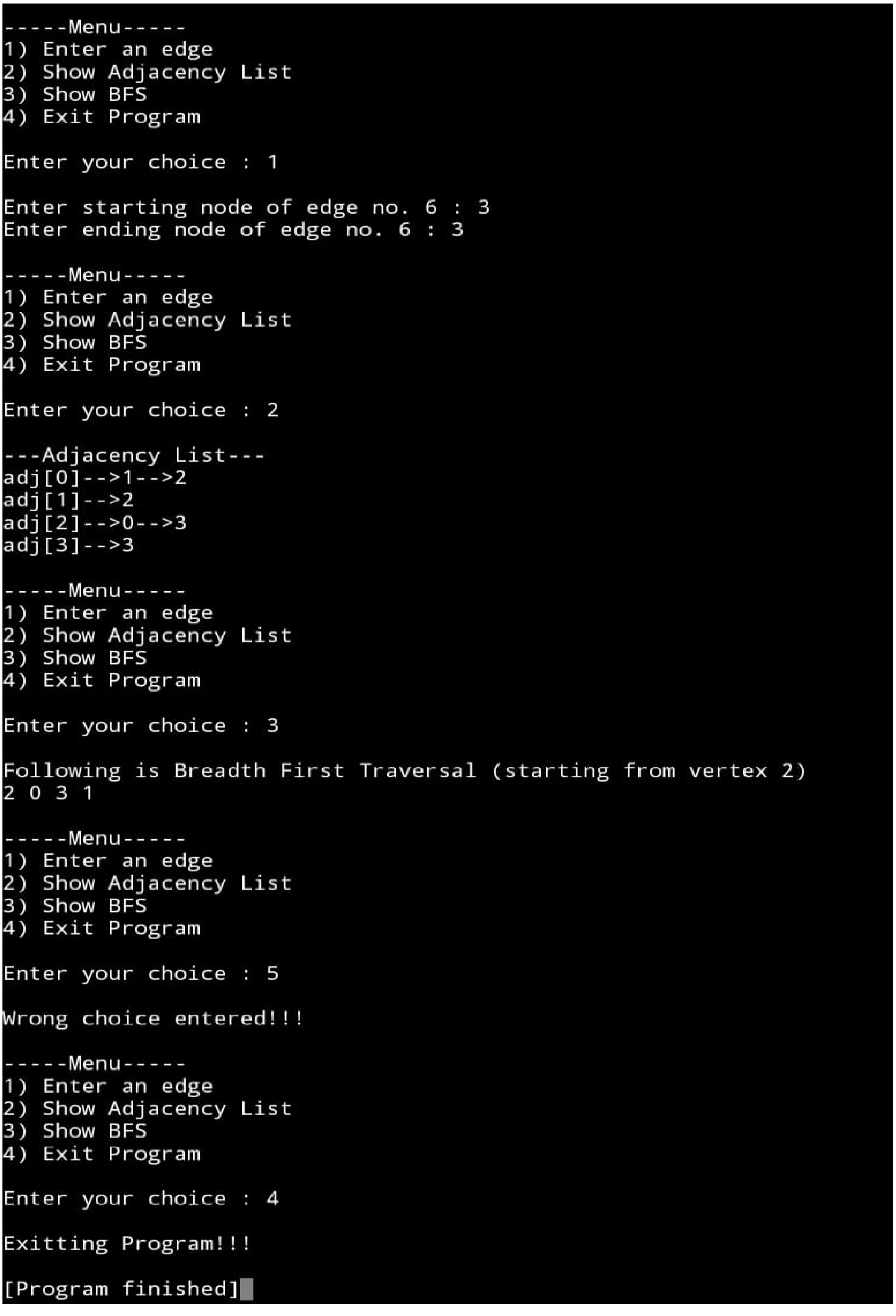
```
		while (!queue.empty())
		{
			// Dequeue a vertex from queue and print it
			s = queue.front();
			cout << s << " ";
			queue.pop_front();

			// Get all adjacent vertices of the dequeued
			// vertex s. If a adjacent has not been visited,
			// then mark it visited and enqueue it
			for (i = adj[s].begin(); i != adj[s].end(); ++i)
			{
				if (!visited[*i])
				{
					visited[*i] = true;
					queue.push_back(*i);
				}
			}
		}
}

// Driver program to test methods of graph class
int main()
{
		int v, i = 1, start, end, c;
		cout << "\n----------BFS----------\n";
		cout << "\nEnter number of vertices : ";
		cin >> v;
		// Create a graph given in the above diagram
		Graph g(v);
		while (1)
		{
			cout << "\n-----Menu-----\n1) Enter an edge\n2) Show Adjacency List\n3) Show BFS\n4) Exit 
Program\n";
			cout << "\nEnter your choice : ";
```

```
        cin >> c;
        switch (c)
        {
        case 1:
            cout << "\nEnter starting node of edge no. " << i << " : ";
            cin >> start;
            cout << "Enter ending node of edge no. " << i << " : ";
            cin >> end;
            i++;
            g.addEdge(start, end);
            break;
        case 2:
            cout << "\n---Adjacency List---" << endl;
            g.print();
            break;
        case 3:
            cout << "\nFollowing is Breadth First Traversal (starting from vertex 2) \n";
            g.BFS(2);
            cout << endl;
            break;
        case 4:
            cout << "\nExitting Program!!!\n";
            exit(0);
        default:
            cout << "\nWrong choice entered!!!\n";
        }
    }
    return 0;
}
```

Output 2:



```
----------BFS----------

Enter number of vertices : 4

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 1 : 0
Enter ending node of edge no. 1 : 1

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 2 : 0
Enter ending node of edge no. 2 : 2

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 3 : 1
Enter ending node of edge no. 3 : 2

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 4 : 2
Enter ending node of edge no. 4 : 0

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 5 : 2
Enter ending node of edge no. 5 : 3
```



```
-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 1

Enter starting node of edge no. 6 : 3
Enter ending node of edge no. 6 : 3

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 2

---Adjacency List---
adj[0]-->1-->2
adj[1]-->2
adj[2]-->0-->3
adj[3]-->3

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 3

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 5

Wrong choice entered!!!

-----Menu-----
1) Enter an edge
2) Show Adjacency List
3) Show BFS
4) Exit Program

Enter your choice : 4

Exitting Program!!!

[Program finished]
```

Conclusion: This program implements graph data structure and performs graph traversals.