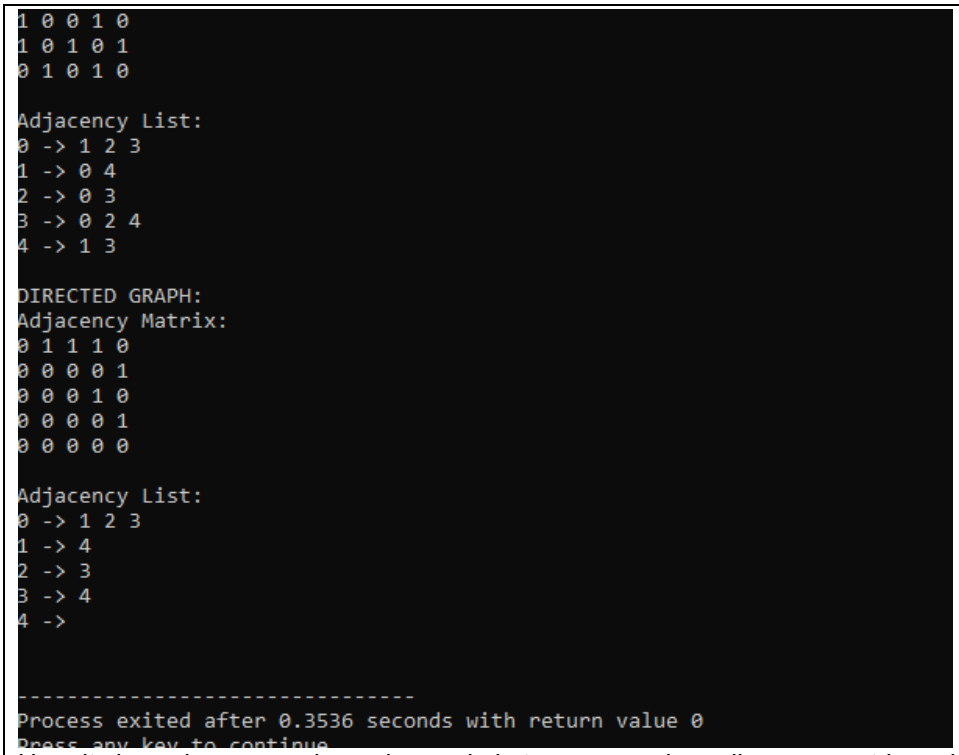
Activity No. <n>	
<Replace with Title>	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09/30/25
Section: CPE 010-CPE21S4	**Date Submitted:**09/30/25
Name(s): Kerwin Jan B. Catungal	**Instructor:** Engr. Roman M. Richard
A. Output(s) and Observation(s)	
ILO A:	

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

class GraphMatrix {
private:
    int vertices;
    vector<vector<int>> adjMatrix;

public:
    GraphMatrix(int v) {
        vertices = v;
        adjMatrix.resize(v, vector<int>(v, 0));
    }

    void addEdge(int u, int v, bool directed = false) {
        adjMatrix[u][v] = 1;
        if (!directed) {
            adjMatrix[v][u] = 1;
        }
    }

    void displayMatrix() {
        cout << "Adjacency Matrix:" << endl;
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

class GraphList {
private:
    int vertices;
    vector<list<int>> adjList;

public:
    GraphList(int v) {
        vertices = v;
        adjList.resize(v);
    }

    void addEdge(int u, int v, bool directed = false) {
        adjList[u].push_back(v);
        if (!directed) {
            adjList[v].push_back(u);
        }
    }

    void displayList() {
        cout << "Adjacency List:" << endl;
        for (int i = 0; i < vertices; i++) {
            cout << i << " -> ";
            for (int v : adjList[i]) {
                cout << v << " ";
            }
            cout << endl;
        }
    }
};

int main() {

    int V = 5;

    cout << "UNDIRECTED GRAPH:" << endl;
    GraphMatrix g1(V);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(1, 4);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    g1.displayMatrix();
    cout << endl;

    GraphList g2(V);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(0, 3);
    g2.addEdge(1, 4);
    g2.addEdge(2, 3);
    g2.addEdge(3, 4);
    g2.displayList();
    cout << endl;

    cout << "DIRECTED GRAPH:" << endl;
    GraphMatrix g3(V);
    g3.addEdge(0, 1, true);
    g3.addEdge(0, 2, true);
    g3.addEdge(0, 3, true);
    g3.addEdge(1, 4, true);
    g3.addEdge(2, 3, true);
    g3.addEdge(3, 4, true);
    g3.displayMatrix();
    cout << endl;

    GraphList g4(V);
    g4.addEdge(0, 1, true);
    g4.addEdge(0, 2, true);
    g4.addEdge(0, 3, true);
    g4.addEdge(1, 4, true);
    g4.addEdge(2, 3, true);
    g4.addEdge(3, 4, true);
    g4.displayList();
    cout << endl;

    return 0;
}
```



Here it shows how a graph can be made in two ways using adjacency matrix and adjacency list. The matrix is like a table that marks if two points are connected, while the list just stores the neighbors of each point. It makes both undirected and directed graphs so you can see the difference. With this, it's easier to compare how the two methods work and when to use them.

ILO B1:

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <set>
#include <map>
using namespace std;


template <typename T>
struct Edge {
    int src;
    int dest;
    T weight;
};


template <typename T>
class Graph {
public:
    Graph(int n) {
        V = n;
    }

    void addEdge(Edge<T> e) {
        if (e.src >= 1 && e.src <= V && e.dest >= 1 && e.dest <= V) {
            edges.push_back(e);
        } else {
            cout << "Vertex out of bounds!" << endl;
        }
    }


    vector<Edge<T>> outgoing(int v) const {
        vector<Edge<T>> out;
        for (auto &e : edges) {
            if (e.src == v) out.push_back(e);
        }
        return out;
    }

    int vertices() const { return V; }


    void printGraph() const {
        for (int i = 1; i <= V; i++) {
            cout << i << ": ";
            auto out = outgoing(i);
            for (auto &e : out) {
                cout << "{" << e.dest << "} ";
            }
            cout << endl;
        }
    }

private:
    int V;
    vector<Edge<T>> edges;
```

```cpp
};

template <typename T>
vector<int> DFS(const Graph<T> &G, int start) {
    stack<int> st;
    vector<int> order;
    set<int> visited;

    st.push(start);
    while (!st.empty()) {
        int curr = st.top();
        st.pop();

        if (visited.find(curr) == visited.end()) {
            visited.insert(curr);
            order.push_back(curr);


            for (auto e : G.outgoing(curr)) {
                if (visited.find(e.dest) == visited.end()) {
                    st.push(e.dest);
                }
            }
        }
    }
    return order;
}

template <typename T>
Graph<T> makeGraph() {
    Graph<T> G(8);
    map<int, vector<int>> edges;
    edges[1] = {2, 5};
    edges[2] = {1, 5, 4};
    edges[3] = {4, 7};
    edges[4] = {2, 3, 5, 6, 8};
    edges[5] = {1, 2, 4, 8};
    edges[6] = {4, 7, 8};
    edges[7] = {3, 6};
    edges[8] = {4, 5, 6};

    for (auto &i : edges) {
        for (auto &j : i.second)
            G.addEdge(Edge<T>(i.first, j, 0));
    }
    return G;
}

int main() {
    Graph<int> G = makeGraph<int>();
    cout << "Graph adjacency list:" << endl;
    G.printGraph();
```

```
Graph adjacency list:
1: {2} {5}
2: {1} {5} {4}
3: {4} {7}
4: {2} {3} {5} {6} {8}
5: {1} {2} {4} {8}
6: {4} {7} {8}
7: {3} {6}
8: {4} {5} {6}

DFS Order:
1 5 8 6 7 3 4 2

--------------------------------
Process exited after 0.3613 seconds with return value 0
Press any key to continue . . .
```

And here it builds a graph and shows its adjacency list, then performs Depth First Search (DFS). The graph is made using edges that connect the vertices, and the adjacency list shows which nodes are connected to each other. The DFS part explores the graph starting from one point and visits all the connected nodes step by step. It uses a stack to keep track of the next nodes to explore. This way, the program shows both the structure of the graph and the order in which nodes are visited.

ILO B2:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <set>
using namespace std;


template <typename T>
struct Edge {
    int src;
    int dest;
    T weight;
};


template <typename T>
class Graph {
public:
    Graph(int n) {
        V = n;
    }

    void addEdge(int u, int v, T w) {
        if (u >= 1 && u <= V && v >= 1 && v <= V) {
            edges.push_back({u, v, w});
        } else {
            cout << "Invalid vertex" << endl;
        }
    }

    vector<Edge<T>> getEdgesFrom(int v) const {
        vector<Edge<T>> result;
        for (auto &e : edges) {
            if (e.src == v) result.push_back(e);
        }
        return result;
    }

    int vertices() const { return V; }


    void printGraph() const {
        for (int i = 1; i <= V; i++) {
            cout << i << ": ";
            auto adj = getEdgesFrom(i);
            for (auto &e : adj) {
                cout << "{" << e.dest << ": " << e.weight << "} ";
            }
            cout << endl;
        }
    }

private:
    int V;
    vector<Edge<T>> edges;
};
```

```cpp
template <typename T>
Graph<T> makeGraph() {
    Graph<T> G(8);

    map<int, vector<pair<int, T>>> data;
    data[1] = {{2, 2}, {5, 3}};
    data[2] = {{1, 2}, {5, 5}, {4, 1}};
    data[3] = {{4, 2}, {7, 3}};
    data[4] = {{2, 1}, {3, 2}, {5, 2}, {6, 4}, {8, 5}};
    data[5] = {{1, 3}, {2, 5}, {4, 2}, {8, 3}};
    data[6] = {{4, 4}, {7, 4}, {8, 1}};
    data[7] = {{3, 3}, {6, 4}};
    data[8] = {{4, 5}, {5, 3}, {6, 1}};

    for (auto &i : data) {
        for (auto &j : i.second) {
            G.addEdge(i.first, j.first, j.second);
        }
    }

    return G;
}

template <typename T>
vector<int> BFS(const Graph<T>& G, int start) {
    queue<int> q;
    set<int> visited;
    vector<int> order;

    q.push(start);

    while (!q.empty()) {
        int curr = q.front();
        q.pop();

        if (visited.find(curr) == visited.end()) {
            visited.insert(curr);
            order.push_back(curr);

            for (auto e : G.getEdgesFrom(curr)) {
                if (visited.find(e.dest) == visited.end()) {
                    q.push(e.dest);
                }
            }
        }
    }
    return order;
}

int main() {
```

```
Graph adjacency list:
1: {2: 2} {5: 3}
2: {1: 2} {5: 5} {4: 1}
3: {4: 2} {7: 3}
4: {2: 1} {3: 2} {5: 2} {6: 4} {8: 5}
5: {1: 3} {2: 5} {4: 2} {8: 3}
6: {4: 4} {7: 4} {8: 1}
7: {3: 3} {6: 4}
8: {4: 5} {5: 3} {6: 1}
BFS Order of vertices:
1
2
5
4
8
3
6
7


--------------------------------
Process exited after 0.3489 seconds with return value 0
Press any key to continue . . .
```

This program makes a graph with 8 vertices and shows it in an adjacency list form with weights. After that, it runs Breadth First Search (BFS), which explores the graph level by level starting from a chosen vertex. It uses a queue to keep track of the next nodes to visit. The program first displays which nodes are connected and their weights, then it prints the order of nodes visited by BFS.

B. Answers to Supplementary Activity

2. **ILO C: Demonstrate an understanding of graph implementation, operations and traversal methods.**

1. A person wants to visit different locations indicated on a map. He starts from one location (vertex) and wants to visit every vertex until it finishes from one vertex, backtracks, and then explore other vertex from same vertex. Discuss which algorithm would be most helpful to accomplish this task.

In my perspective, Depth-First Search (DFS) is the most appropriate algorithm for this task. DFS begins at a selected vertex and explores each path as deeply as possible before backtracking. This allows the algorithm to thoroughly traverse each route one at a time. It's particularly effective when the goal is to explore all possible paths from a specific starting point. Once one path is fully explored, DFS returns to the previous node to continue exploring any remaining unvisited paths.

2. Identify the equivalent of DFS in traversal strategies for trees. To efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.

In tree data structures, pre-order traversal is the counterpart of Depth-First Search (DFS). Both methods follow a depth-first strategy, where each node is visited before its subtrees are explored. Specifically, pre-order traversal adheres to the Root → Left → Right order: it processes the root node first, then traverses the left subtree, followed by the right. This approach ensures that every node is examined before delving deeper into its branches. Similar to DFS, pre-order traversal can be implemented either recursively or using an explicit stack, and it plays an essential role in understanding and navigating tree structures.

3. In the performed code, what data structure is used to implement the Breadth First Search?

Breadth-First Search (BFS) is implemented using a queue, which operates on the First-In, First-Out (FIFO) principle. As nodes are discovered, they are enqueued, and each node is processed in the same order it was added. This approach allows BFS to explore nodes level by level, ensuring that all nodes at the current depth are visited before moving on to the next level.

4. How many times can a node be visited in the BFS?

In Breadth-First Search (BFS), each node is visited only once. Once a node is marked as visited, it is not added to the queue again, which avoids redundant processing and improves efficiency. This one-time visitation is especially important in unweighted graphs, as it ensures that the shortest path to each node is found. To keep track of explored nodes, a visited list or set is commonly used.

C. Conclusion & Lessons Learned

From this lab activity, I had a much better understanding of the differences and appropriate situations for using Depth-First Search (DFS) and Breadth-First Search (BFS). DFS is suitable for deep-path exploration in a graph, while BFS offers level-wise traversal. This has made me appreciate even more how important coding practical applications are for theoretical concepts as it helped me visualize in a stepwise manner how each of the traversal algorithms proceeds. Looking ahead, I trust further coding practice-among all other preparations-will become a necessity for developing all such practical skills through more complex graphs and edge cases.

D. Assessment Rubric

E. External References