Activity No. <10>	
GRAPHS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/30/25**
Section: CPE21S4	**Date Submitted: 9/30/25**
Name(s): QUIOYO, ANGELO	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO_A:
Main Cpp:

A.cpp | B1.cpp | B2.cpp

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

class AdjacencyMatrixGraph {
private:
    int vertexCount;
    vector<vector<int>> matrix;

public:
    AdjacencyMatrixGraph(int v) {
        vertexCount = v;
        matrix.resize(v, vector<int>(v, 0));
    }

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

    void printMatrix() {
        cout << "Adjacency Matrix Representation:" << endl;
        for (int i = 0; i < vertexCount; i++) {
            for (int j = 0; j < vertexCount; j++) {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

// ====================================
// Graph Representation using Adjacency list
// ====================================
class AdjacencyListGraph {
private:
    int vertexCount;
    vector<list<int>> listAdj;

public:
    AdjacencyListGraph(int v) {
        vertexCount = v;
        listAdj.resize(v);
    }

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

    void printList() {
        cout << "Adjacency List Representation:" << endl;
        for (int i = 0; i < vertexCount; i++) {
            cout << i << " -> ";
            for (int v : listAdj[i]) {
                cout << v << " ";
            }
            cout << endl;
        }
    }
};


// ========================
// Main Program
// ========================
int main() {
    int V = 5;

    // Undirected Graph
    cout << "UNDIRECTED GRAPH:" << endl;
    AdjacencyMatrixGraph matrixGraph1(V);
    matrixGraph1.addEdge(0, 1); // A-B
    matrixGraph1.addEdge(0, 2); // A-C
    matrixGraph1.addEdge(0, 3); // A-D
    matrixGraph1.addEdge(1, 4); // B-E
    matrixGraph1.addEdge(2, 3); // C-D
    matrixGraph1.addEdge(3, 4); // D-E
    matrixGraph1.printMatrix();
    cout << endl;

    AdjacencyListGraph listGraph1(V);
    listGraph1.addEdge(0, 1);
    listGraph1.addEdge(0, 2);
    listGraph1.addEdge(0, 3);
    listGraph1.addEdge(1, 4);
    listGraph1.addEdge(2, 3);
    listGraph1.addEdge(3, 4);
    listGraph1.printList();
    cout << endl;

    // Directed Graph
    cout << "DIRECTED GRAPH:" << endl;
    AdjacencyMatrixGraph matrixGraph2(V);
    matrixGraph2.addEdge(0, 1, 1, true); // A->B
    matrixGraph2.addEdge(0, 2, 3, true); // A->C
    matrixGraph2.addEdge(0, 3, 4, true); // A->D
    matrixGraph2.addEdge(1, 4, 2, true); // B->E
    matrixGraph2.addEdge(2, 3, 1, true); // C->D
    matrixGraph2.addEdge(3, 4, 5, true); // D->E
    matrixGraph2.printMatrix();
    cout << endl;

    AdjacencyListGraph listGraph2(V);
    listGraph2.addEdge(0, 1, true);
    listGraph2.addEdge(0, 2, true);
    listGraph2.addEdge(0, 3, true);
    listGraph2.addEdge(1, 4, true);
    listGraph2.addEdge(2, 3, true);
    listGraph2.addEdge(3, 4, true);
    listGraph2.printList();
    cout << endl;

    return 0;
}
```

Output:

```
UNDIRECTED GRAPH:
Adjacency Matrix Representation:
0 1 1 1 0
1 0 0 0 1
1 0 0 1 0
1 0 1 0 1
0 1 0 1 0

Adjacency List Representation:
0 -> 1 2 3
1 -> 0 4
2 -> 0 3
3 -> 0 2 4
4 -> 1 3

DIRECTED GRAPH:
Adjacency Matrix Representation:
0 1 3 4 0
0 0 0 0 2
0 0 0 1 0
0 0 0 0 5
0 0 0 0 0

Adjacency List Representation:
0 -> 1 2 3
1 -> 4
2 -> 3
3 -> 4
4 ->


--------------------------------
Process exited after 0.1927 seconds with return value 0
Press any key to continue . . .
```

Explanation:
This code defines two types of graph representations: an adjacency matrix and an adjacency list. The GraphMatrix class uses a 2D matrix to store connections between vertices, and the GraphList class uses lists to store adjacent vertices. The addEdge method adds edges to the graph, allowing for both directed and undirected graphs. The displayMatrix and displayList methods print the graph in matrix and list formats, respectively. The program demonstrates how to create graphs, add edges, and display them in both matrix and list formats.

ILO_B1:

Main CPP:

A.cpp | B1.cpp | B2.cpp

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

struct Edge {
    int source;
    int destination;
    int weight;
};

class Graph {
private:
    int vertexCount;
    vector<Edge> edges;

public:
    Graph(int vertices) {
        vertexCount = vertices;
    }

    void addEdge(int u, int v, int w = 0) {
        if (u >= 1 && u <= vertexCount && v >= 1 && v <= vertexCount) {
            edges.push_back({u, v, w});
        }
    }

    vector<Edge> getOutgoingEdges(int u) const {
        vector<Edge> result;
        for (const auto& edge : edges) {
            if (edge.source == u) {
                result.push_back(edge);
            }
        }
        return result;
    }

    void displayGraph() const {
        for (int i = 1; i <= vertexCount; i++) {
            cout << i << ": ";
            for (const auto& edge : edges) {
                if (edge.source == i) {
                    cout << "{" << edge.destination << "} ";
                }
            }
            cout << endl;
        }
    }
};

```

```cpp
vector<int> depthFirstSearch(const Graph& graph, int start) {
    stack<int> stk;
    set<int> visitedNodes;
    vector<int> traversalOrder;

    stk.push(start);
    while (!stk.empty()) {
        int node = stk.top();
        stk.pop();
        if (visitedNodes.find(node) == visitedNodes.end()) {
            visitedNodes.insert(node);
            traversalOrder.push_back(node);
            for (const auto& edge : graph.getOutgoingEdges(node)) {
                if (visitedNodes.find(edge.destination) == visitedNodes.end()) {
                    stk.push(edge.destination);
                }
            }
        }
    }
    return traversalOrder;
}

Graph createSampleGraph() {
    Graph g(8);

    map<int, vector<int>> adjacencyList = {
        {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}}
    };

    for (const auto& [u, neighbors] : adjacencyList) {
        for (const auto& v : neighbors) {
            g.addEdge(u, v);
        }
    }

    return g;
}

int main() {
    Graph graph = createSampleGraph();

    cout << "Graph's Adjacency List:" << endl;
    graph.displayGraph();
    cout << endl;

    cout << "DFS Traversal Order:" << endl;
    vector<int> dfsOrder = depthFirstSearch(graph, 1);
    for (int v : dfsOrder) {
        cout << v << " ";
    }
    cout << endl;

    return 0;
}

```

Output:

```
Graph's 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 Traversal Order:
1 5 8 6 7 3 4 2

--------------------------------
Process exited after 0.1172 seconds with return value 0
Press any key to continue . . .
```

Explanation:

This code defines a Graph class that represents a graph using an edge list where each edge has a source, destination, and weight. It includes a DFS (Depth-First Search) function, which uses a stack to explore all vertices, visiting neighbors in a depth-first manner. The graph is created using a set of predefined edges with weights, and the traversal starts from a specified vertex. The displayGraph method outputs the adjacency list of the graph. Finally, the program performs a DFS traversal and prints the order of visited vertices.

ILO B2:

Main CPP:

A.cpp | B1.cpp | B2.cpp

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

// Edge structure template
template <typename T>
struct Edge {
    size_t source;
    size_t destination;
    T weight;

    // Comparison operators for sorting based on edge weight
    bool operator<(const Edge<T>& e) const {
        return weight < e.weight;
    }
    bool operator>(const Edge<T>& e) const {
        return weight > e.weight;
    }
};

// Graph class template
template <typename T>
class Graph {
public:
    Graph(size_t vertices) : vertexCount(vertices) {}

    // Add an edge to the graph
    void add_edge(size_t u, size_t v, T w) {
        if (u >= 1 && u <= vertexCount && v >= 1 && v <= vertexCount) {
            edges.push_back({u, v, w});
        } else {
            cerr << "Invalid edge!" << endl;
        }
    }

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

    size_t vertices() const {
        return vertexCount;
    }

    void display() const {
        for (size_t i = 1; i <= vertexCount; ++i) {
            cout << i << " -> ";
            auto adj = outgoing_edges(i);
            for (const auto& e : adj) {
                cout << "{" << e.destination << ":" << e.weight << "} ";
            }
            cout << endl;
        }
    }

private:
    size_t vertexCount;
    vector<Edge<T>> edges;
};

template <typename T>
Graph<T> create_graph() {
    Graph<T> G(8);

    map<int, vector<pair<int, T>>> adjacencyList = {
        {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}}}
    };

    for (const auto& entry : adjacencyList) {
        for (const auto& neighbor : entry.second) {
            G.add_edge(entry.first, neighbor.first, neighbor.second);
        }
    }

    return G;
}

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

    q.push(start);
    while (!q.empty()) {
        size_t current = q.front();
        q.pop();
        if (visited.find(current) == visited.end()) {
            visited.insert(current);
            order.push_back(current);
            for (const auto& e : G.outgoing_edges(current)) {
                if (visited.find(e.destination) == visited.end()) {
                    q.push(e.destination);
                }
            }
        }
    }

    return order;
}

template <typename T>
void run_BFS() {
    auto G = create_graph<T>();

    cout << "Graph adjacency list:" << endl;
    G.display();

    cout << "BFS traversal order of vertices:" << endl;
    auto order = breadth_first_search(G, 1);
    for (auto v : order) {
        cout << v << endl;
    }
}

int main() {
    run_BFS<unsigned>();
    return 0;
}
```

Output:

```
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 traversal order of vertices:
1
2
5
4
8
3
6
7


--------------------------------
Process exited after 0.1247 seconds with return value 0
Press any key to continue . . .
```

Explanation:

This code represents a graph with weighted edges using an edge list, where each edge connects a source and destination with a weight. The Graph class includes a breadth_first_search (BFS) function that explores the graph level by level using a queue to visit each vertex in breadth-first order. It also uses a map to define the graph's edges with weights and a display method to show the adjacency list. The add_edge function adds an edge between two vertices with a weight. The program demonstrates BFS traversal starting from a specific vertex and prints the visited vertices in order.

B. Answers to Supplementary Activity:

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.**
   - **The best algorithm for this problem is Depth First Search (DFS). In DFS, the person start from one vertex and keep going deeper to next connected vertex. If he reach a vertex with no more new path, he backtrack to the last vertex and continue other path. This is same like exploring one road fully before moving to another road. DFS is helpful because it visit every location and make sure nothing is missed.**

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 trees, the equal of DFS is Inorder, Preorder, and Postorder traversal. This is because they also go deep into the tree before moving back. For example, in preorder, you visit root first then go left then right, which is depth style. In graphical view, DFS in graph look like tree traversal because both go down one branch first. So we can say tree traversal and DFS are similar idea but just different structure.**

3. **In the performed code, what data structure is used to implement the Breadth First Search?**
   - **In BFS, the data structure use is a queue. A queue work in first-in-first-out style, so the first node added will be visited first. This is good for BFS because BFS visit level by level. When we push neighbors into queue, they wait until it is their turn. That is why BFS always use queue for implementation.**

4. **How many times can a node be visited in the BFS?**
   - **In BFS, a node is visited only one time. This is because when a node is visited, it is already marked as visited. So BFS not go back to visit it again. If not mark, BFS will loop many times in same node which is wrong. That is why each node can only be visited once in BFS**

C. Conclusion & Lessons Learned: In conclusion, graph traversal is very important to visit every node in a graph or tree. DFS is good for going deep in one path before moving to another path, while BFS is good for going level by level. Both methods are useful but they work in different way. The data structure used in BFS is queue, and each node is only visited once. By learning these, we can understand better how computer explore maps, trees, and networks.

D. Assessment Rubric

E. External References