Activity No. 10.1	
Graphs	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 11 / 13 / 24**
Section: CpE21S4	**Date Submitted: 11 / 13 / 24**
Name(s): Esteban, Fernandez, Sanchez, Tandayu, Valleser	**Instructor:** Engr. Ma. Rizette Sayo

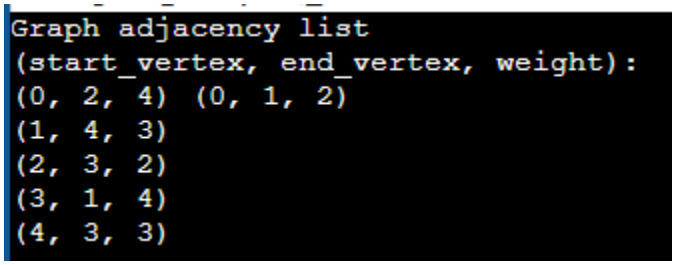
A. Output(s) and Observation(s)

A1 - A3

Code

```
#include <iostream>

// Stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};

// Structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};

class DiaGraph {
    // Insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head) {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;
        newNode->next = head; // Point new node to current head
        return newNode;
    }

    int N; // Number of nodes in the graph

public:
    adjNode** head; // Adjacency list as array of pointers

    // Constructor
    DiaGraph(graphEdge edges[], int n, int N) {
        // Allocate new node
        head = new adjNode*[N]();
        this->N = N;
        // Initialize head pointer for all vertices
        for (int i = 0; i < N; ++i)
            head[i] = nullptr;
        // Construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++) {
```

```
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;
            // Insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver, weight, head[start_ver]);
            // Point head pointer to new node
            head[start_ver] = newNode;
        }
    }

    // Destructor
    ~DiaGraph() {
        for (int i = 0; i < N; i++) {
            adjNode* current = head[i];
            while (current) {
                adjNode* temp = current;
                current = current->next;
                delete temp; // Free each node
            }
        }
        delete[] head; // Free the head array
    }
};

// Print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i) {
    while (ptr != nullptr) {
        std::cout << "(" << i << ", " << ptr->val << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    std::cout << std::endl;
}

// Graph implementation
int main() {
    // Graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0, 1, 2}, {0, 2, 4}, {1, 4, 3}, {2, 3, 2}, {3, 1, 4}, {4, 3, 3}
    };
    int N = 5; // Number of vertices in the graph
    // Calculate number of edges
    int n = sizeof(edges) / sizeof(edges[0]);
    // Construct graph
    DiaGraph diagraph(edges, n, N);
    // Print adjacency list representation of graph
    std::cout << "Graph adjacency list " << std::endl << "(start_vertex, end_vertex, weight):"
<< std::endl;
    for (int i = 0; i < N; i++) {
        // Display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
```

	} return 0; }
Output	

B1

Code	#include <string>
	#include <vector>
	#include <iostream>
	#include <set>
	#include <map>
	#include <stack>
	
	// Forward declaration for the Graph class
	template <typename T>
	class Graph;
	
	// Edge structure representing a weighted edge
	template <typename T>
	struct Edge {
	size_t src;
	size_t dest;
	T weight;
	
	// To compare edges, only compare their weights
	inline bool operator<(const Edge<T> &e) const {
	return this->weight < e.weight;
	}
	inline bool operator>(const Edge<T> &e) const {
	return this->weight > e.weight;
	}
	};
	
	// Overload the << operator for Graph
	template <typename T>
	std::ostream &operator<<(std::ostream &os, const Graph<T> &G);
	
	// Graph class with an edge list representation
	template <typename T>
	class Graph {
	public:
	// Initialize the graph with N vertices

```
    Graph(size_t N) : V(N) {}

    // Return number of vertices in the graph
    auto vertices() const {
        return V;
    }

    // Return all edges in the graph
    const auto &edges() const {
        return edge_list;
    }

    // Add an edge to the graph
    void add_edge(Edge<T> &&e) {
        // Check if the source and destination vertices are within range
        if (e.src >= 1 && e.src <= V &&
            e.dest >= 1 && e.dest <= V) {
            edge_list.emplace_back(e);
        } else {
            std::cerr << "Vertex out of bounds" << std::endl;
        }
    }

    // Returns all outgoing edges from vertex v
    auto outgoing_edges(size_t v) const {
        std::vector<Edge<T>> edges_from_v;
        for (const auto &e : edge_list) {
            if (e.src == v) {
                edges_from_v.emplace_back(e);
            }
        }
        return edges_from_v;
    }

    // Declare friend function to overload << operator
    template <typename U>
    friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);

private:
    size_t V; // Stores number of vertices in graph
    std::vector<Edge<T>> edge_list;
};

// Overload the << operator to output graph structure
template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G) {
    for (size_t i = 1; i <= G.vertices(); ++i) {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (const auto &e : edges) {
            os << "{" << e.dest << ": " << e.weight << "}, ";
```

```
        }
        os << std::endl;
    }
    return os;
}

// Perform DFS on the graph starting from vertex 1
template <typename T>
auto depth_first_search(const Graph<T> &G, size_t start_vertex = 1) {
    std::stack<size_t> stack;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    stack.push(start_vertex);

    while (!stack.empty()) {
        auto current_vertex = stack.top();
        stack.pop();

        // If the current vertex hasn't been visited yet
        if (visited.find(current_vertex) == visited.end()) {
            visited.insert(current_vertex);
            visit_order.push_back(current_vertex);

            for (auto e : G.outgoing_edges(current_vertex)) {
                // If the vertex hasn't been visited, insert it in the stack
                if (visited.find(e.dest) == visited.end()) {
                    stack.push(e.dest);
                }
            }
        }
    }
    return visit_order;
}

// Create a reference graph with predefined edges
template <typename T>
auto create_reference_graph() {
    Graph<T> G(9);
    std::map<size_t, std::vector<std::pair<size_t, T>>> edges;
    edges[1] = {{2, 0}, {5, 0}};
    edges[2] = {{1, 0}, {5, 0}, {4, 0}};
    edges[3] = {{4, 0}, {7, 0}};
    edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 0}};
    edges[5] = {{1, 0}, {2, 0}, {4, 0}, {8, 0}};
    edges[6] = {{4, 0}, {7, 0}, {8, 0}};
    edges[7] = {{3, 0}, {6, 0}};
    edges[8] = {{4, 0}, {5, 0}, {6, 0}};

    for (const auto &i : edges) {
        for (const auto &j : i.second) {
```

	G.add_edge(Edge<T>{i.first, j.first, j.second});
	}
	}
	return G;
	}
	
	// Test DFS function
	template <typename T>
	void test_DFS() {
	// Create an instance of the graph and print it
	auto G = create_reference_graph<T>();
	std::cout << G << std::endl;
	
	// Run DFS starting from vertex ID 1 and print the order
	// in which vertices are visited.
	std::cout << "DFS Order of vertices: " << std::endl;
	auto dfs_visit_order = depth_first_search(G, 1);
	for (auto v : dfs_visit_order) {
	std::cout << v << " ";
	}
	std::cout << std::endl;
	}
	
	// Main function
	int main() {
	using T = unsigned;
	test_DFS<T>();
	return 0;
	}

Output	1: {2: 0}, {5: 0}, 2: {1: 0}, {5: 0}, {4: 0}, 3: {4: 0}, {7: 0}, 4: {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0}, 5: {1: 0}, {2: 0}, {4: 0}, {8: 0}, 6: {4: 0}, {7: 0}, {8: 0}, 7: {3: 0}, {6: 0}, 8: {4: 0}, {5: 0}, {6: 0}, 9: DFS Order of vertices: 1 5 8 6 7 3 4 2

B2

Code	#include <string> #include <vector> #include <iostream> #include <set> #include <map> #include <queue> template <typename T> class Graph;

```
template <typename T>
struct Edge
{
    size_t src;
    size_t dest;
    T weight;

    inline bool operator<(const Edge<T> &e) const
    {
        return this->weight < e.weight;
    }
    inline bool operator>(const Edge<T> &e) const
    {
        return this->weight > e.weight;
    }
};

template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i <= G.vertices(); i++)  // Changed the loop bounds to include the last
vertex
    {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
        {
            os << "{" << e.dest << ": " << e.weight << "}, ";
        }
        os << std::endl;
    }
    return os; // Fixed missing return
}

template <typename T>
class Graph
{
public:
    Graph(size_t N) : V(N) {}

    auto vertices() const
    {
        return V;
    }

    auto &edges() const
    {
        return edge_list;
    }

    void add_edge(Edge<T> &&e)
```

```
    {
        if (e.src >= 1 && e.src <= V &&
            e.dest >= 1 && e.dest <= V)
        {
            edge_list.emplace_back(e);
        }
        else
        {
            std::cerr << "Vertex out of bounds" << std::endl;
        }
    }

    auto outgoing_edges(size_t v) const
    {
        std::vector<Edge<T>> edges_from_v;
        for (auto &e : edge_list)
        {
            if (e.src == v)
            {
                edges_from_v.emplace_back(e);
            }
        }
        return edges_from_v;
    }

    template <typename U>
    friend std::ostream &operator<<(std::ostream &os, const Graph<U> &G);

private:
    size_t V;
    std::vector<Edge<T>> edge_list;
};

// Create a graph for testing
template <typename T>
auto create_reference_graph()
{
    Graph<T> G(8);  // 8 vertices
    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;

    // Define edges as {source, {destination, weight}}
    edges[1] = {{2, 2}, {5, 3}};
    edges[2] = {{1, 2}, {5, 5}, {4, 1}};
    edges[3] = {{4, 2}, {7, 3}};
    edges[4] = {{2, 1}, {3, 2}, {5, 2}, {6, 4}, {8, 5}};
    edges[5] = {{1, 3}, {2, 5}, {4, 2}, {8, 3}};
    edges[6] = {{4, 4}, {7, 4}, {8, 1}};
    edges[7] = {{3, 3}, {6, 4}};
    edges[8] = {{4, 5}, {5, 3}, {6, 1}};

    // Add edges to the graph
```

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

    return G;
}

// Breadth-First Search (BFS)
template <typename T>
auto breadth_first_search(const Graph<T> &G, size_t start)
{
    std::queue<size_t> queue;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    queue.push(start);  // BFS always starts from vertex ID 1
    while (!queue.empty())
    {
        auto current_vertex = queue.front();
        queue.pop();

        // If the current vertex hasn't been visited in the past
        if (visited.find(current_vertex) == visited.end())
        {
            visited.insert(current_vertex);
            visit_order.push_back(current_vertex);

            // Add unvisited neighbors to the queue
            for (auto e : G.outgoing_edges(current_vertex))
            {
                if (visited.find(e.dest) == visited.end())
                {
                    queue.push(e.dest);
                }
            }
        }
    }
    return visit_order;
}

// Test BFS on the reference graph
template <typename T>
void test_BFS()
{
    // Create an instance of and print the graph
    auto G = create_reference_graph<T>();
    std::cout << G << std::endl;
```

```
    // Run BFS starting from vertex ID 1 and print the order in which vertices are visited
    std::cout << "BFS Order of vertices: " << std::endl;
    auto bfs_visit_order = breadth_first_search(G, 1);
    for (auto v : bfs_visit_order)
    {
        std::cout << v << std::endl;
    }
}

int main()
{
    using T = unsigned;
    test_BFS<T>();
    return 0;
}
```

Output

```
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
```

B. Answers to Supplementary Activity

1. The most suitable algorithm for visiting all vertices in the given scenario is **Depth-First Search (DFS)**. DFS explores as far as possible along each branch before backtracking, ensuring that all vertices are visited. It efficiently handles revisiting and exploring other vertices from the same starting point.

2. In tree traversal, the **Depth-First Search (DFS)** corresponds to three strategies: **Pre-order**, **In-order**, and **Post-order**. These traversals explore the tree by going as deep as possible down one branch before backtracking to explore others. Each has a different order of visiting nodes: Pre-order visits the node first, In-order visits the node between its subtrees, and Post-order visits the node last.

   Pseudocode for Pre-order (example):

   a. def pre_order(node):

b. if node: print(node.data)

c. pre_order(node.left)

d. pre_order(node.right)

3. In the provided Breadth-First Search (BFS) code, the primary data structure used is a queue. The queue follows the FIFO (First In, First Out) principle, which is essential for BFS because it ensures that nodes are explored level by level, starting from the root and moving outward. Nodes are enqueued when visited and dequeued to process their neighbors in the correct order.

4. In **Breadth-First Search (BFS)**, each node can be visited **at most once**. BFS explores each node in a level-order fashion, meaning it processes each node when it is dequeued from the queue. Once a node is visited, it is marked as visited (usually in a set or boolean array), preventing it from being added to the queue or visited again. This ensures that no node is processed more than once during the traversal.

C. Conclusion & Lessons Learned

Summary of Lessons Learned

Through this lesson, we explored **graph traversal algorithms**—specifically **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. We learned how DFS explores a graph deeply, visiting nodes and backtracking when necessary, while BFS explores nodes level by level using a queue. We also examined the conditions under which each algorithm is best suited, and understood how the data structures (such as stacks for DFS and queues for BFS) support the traversal process.

Analysis of the Procedure

The procedure for learning these algorithms involved understanding their conceptual differences, implementing the algorithms using simple pseudocode, and exploring how they operate on graphs and trees. By examining the mechanics of DFS and BFS, we were able to differentiate their traversal strategies, like DFS's depth-first exploration versus BFS's level-order traversal. Both algorithms are crucial for various applications like searching, pathfinding, and traversal in both directed and undirected graphs.

Analysis of the Supplementary Activity

The supplementary activity provided an opportunity to implement BFS and DFS in code, test the algorithms, and observe their behaviors. This hands-on practice solidified our understanding of the theoretical concepts, as it demonstrated how different data structures (like stacks and queues) are essential for the traversal process. This activity also helped clarify edge cases such as handling nodes that are already visited in BFS, ensuring each node is visited at most once.

Concluding Statement / Feedback

I believe I performed well in this activity, successfully grasping the core principles of BFS and DFS, and implementing them with the appropriate data structures. However, I recognize an area for improvement in understanding more complex graph traversal problems (e.g., cycles, weighted graphs) and how advanced variations of these algorithms can be applied. More practice with edge cases and performance optimization would also be beneficial.

D. Assessment Rubric

E. External References