Activity No. 10	
Graphs	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11-13-2024
Section: CPE21S4	**Date Submitted:** 11-13-2024
Name(s): BONIFACIO, NYKO ADREIN MAGISTRADO, AIRA PAULEEN PLANTA, CALVIN EARL SOLIS, PAUL VINCENT VIRTUCIO, DOMINIC	**Instructor:** Prof. Sayo

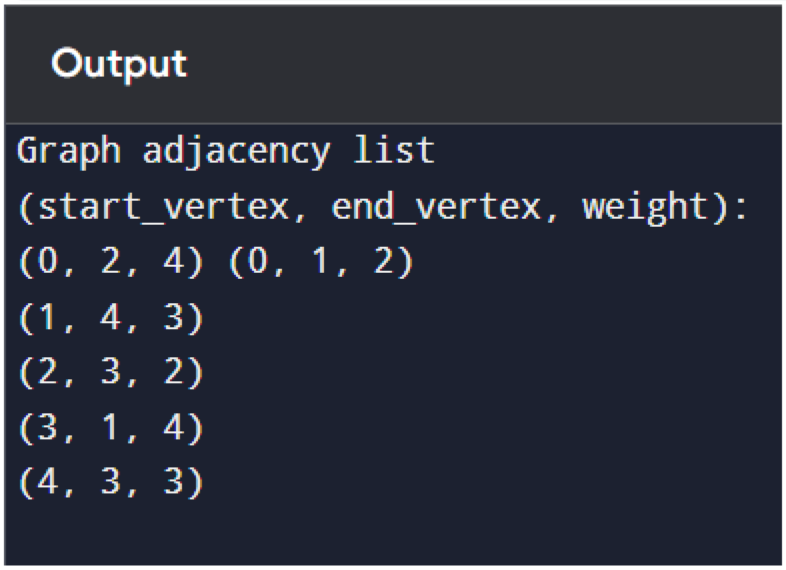
6. Output(s) and Observation(s)

ILO A

Screenshot + Code Output

C/C++

```cpp
#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++)
            delete[] head[i];
        delete[] head;
    }
};

// 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 = 6; // 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
Graph adjacency list
(start_vertex, end_vertex, weight):
(0, 2, 4) (0, 1, 2)
(1, 4, 3)
(2, 3, 2)
(3, 1, 4)
(4, 3, 3)
```

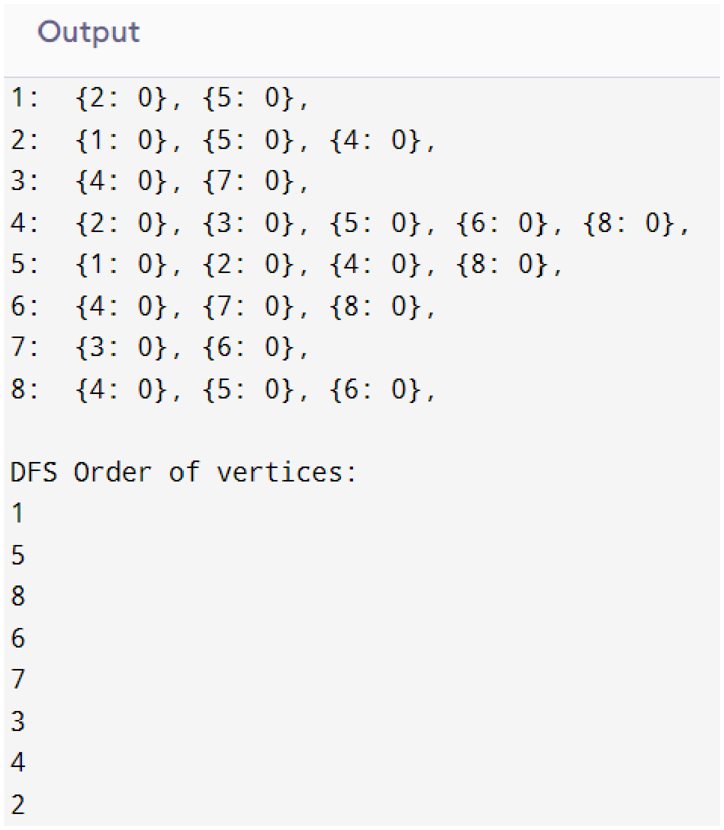
Observations	Implementation is efficient, handles memory management well, and is easy to extend for further graph algorithms like depth-first search (DFS) nad/or breadth-first search (BFS)

ILO B

Screenshot + Code Output

```
C/C++
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <stack>

template <typename T>
class Graph;

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

    // To compare edges, only compare their weights,
    // and not the source/destination vertices
    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;
    }
};

// Define the << operator for the Graph class outside of Edge
template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i < G.vertices(); i++)
    {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
            os << "{" << e.dest << ": " << e.weight << "}, ";
        os << std::endl;
    }
    return os;
}

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
    auto &edges() const
    {
        return edge_list;
    }

    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 (auto &e : edge_list)
        {
            if (e.src == v)
                edges_from_v.emplace_back(e);
        }
        return edges_from_v;
    }

    // Overloads the << operator so a graph can be written
directly to a stream
    // Can be used as std::cout << obj << std::endl;
    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;
};

template <typename T>
auto depth_first_search(const Graph<T> &G, size_t start)
{
    std::stack<size_t> stack;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    stack.push(start); // DFS starts from the given vertex

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

            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())
                {
```

```cpp
                    stack.push(e.dest);
                }
            }
        }
    }

    return visit_order;
}

template <typename T>
auto create_reference_graph()
{
    Graph<T> G(9);
    std::map<unsigned, 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 (auto &i : edges)
        for (auto &j : i.second)
            G.add_edge(Edge<T>{i.first, j.first, j.second});

    return G;
}

template <typename T>
void test_DFS()
{
    // Create an instance of and print the graph
    auto G = create_reference_graph<unsigned>();
    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::endl;
}

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

	
Observations	Provides a good starting point for graph-based algorithms. It’s modular, flexible, and implements core concepts like DFS and graph creation. However, for larger-scale applications, the current edge list representation and DFS implementation could be optimized.

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},

DFS Order of vertices:
1
5
8
6
7
3
4
2
```

B.1. Depth - First Search

Screenshot + Code Output	

C/C++

```cpp
#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;
```

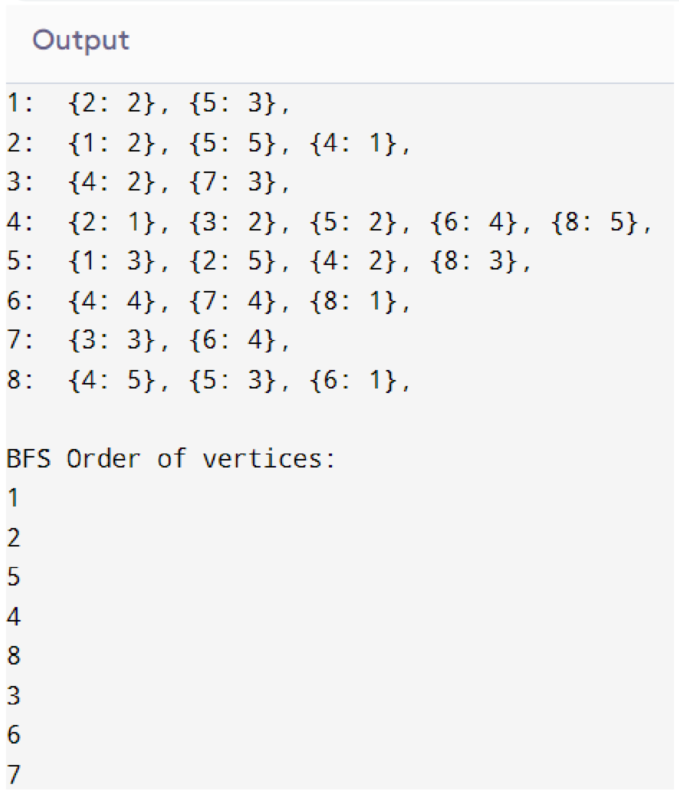
	(code below)

```
    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;
    }
};

// Moved the << operator to associate it with the Graph class
template <typename T>
class Graph
{
public:
    // Initialize the graph with N vertices
    Graph(size_t N) : V(N) {}

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

    // Return all edges in the graph
    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 (auto &e : edge_list)
        {
```

	(code below)

```cpp
            if (e.src == v)
            {
                edges_from_v.emplace_back(e);
            }
        }
        return edges_from_v;
    }

    // Overload the << operator to print the graph
    template <typename U>
    friend std::ostream &operator<<(std::ostream &os, const
Graph<U> &G);

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

// Overload the << operator to print the graph's adjacency list
template <typename T>
std::ostream &operator<<(std::ostream &os, const Graph<T> &G)
{
    for (auto i = 1; i <= G.vertices(); i++) // Fixed loop to go
through all vertices
    {
        os << i << ":\t";
        auto edges = G.outgoing_edges(i);
        for (auto &e : edges)
        {
            os << "{" << e.dest << ": " << e.weight << "}, ";
        }
        os << std::endl;
    }
    return os; // Return the stream to allow chaining
}

// Helper function to create a reference graph for testing
template <typename T>
auto create_reference_graph()
{
    Graph<T> G(8); // Graph has 8 vertices

    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
    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}};

    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 function
template <typename T>
auto breadth_first_search(const Graph<T> &G)
{
    std::queue<size_t> queue;
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    queue.push(1); // Start BFS from vertex ID 1

    while (!queue.empty())
    {
        auto current_vertex = queue.front();
        queue.pop();

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

            // Push all unvisited neighbors onto 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 function to run BFS and print the graph
template <typename T>
void test_BFS()
{
    // Create an instance of the graph and print it
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;

    // Run BFS starting from vertex ID 1 and print the visit
order
    std::cout << "BFS Order of vertices: " << std::endl;
    auto bfs_visit_order = breadth_first_search(G);
    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
```

Observations	Successfully implements a basic graph class with an edge list representation and BFS traversal. Well-structured, but there are opportunities for improving both the performance and functionality. Switching to more efficient graph representations (e.g., adjacency lists) and considering weighted traversals would make the code more versatile and scalable for larger graphs.

B.2. Breadth - First Search

7. Answers to Supplementary Activity

1. Since Depth-First Search (DFS) lets you explore really deep along one path before going back and checking out other options, it's the best way to visit different spots on a map starting from one place and then backtracking to see all the other places.
2. Pre-order, in-order, and post-order traversals are the tree traversal methods' equivalents of DFS. Each method visits nodes in a certain order: post-order visits the children before the root, in-order visits the left child before the root, and pre-order visits the root first.
3. The order of vertex exploration in the Breadth-First Search (BFS) implementation is controlled by a queue data structure, which enables the methodical study of every nearby vertex before delving further into the graph.

4. Because a node can only be visited once in Breadth-First Search (BFS), every vertex is processed effectively without having to be revisited.

8. Conclusion & Lessons Learned

In conclusion, we learned about implementations of graphs in C++ through this activity. This exercise introduced graph data structures in depth, examining the details of adjacency matrices and lists with specific advantages and disadvantages. We learned a number of algorithms, even Depth-First Search and Breadth-First Search, understanding at which points to apply them along with their usage in C++. All this enabled us to construct, modify, and traverse graphs by adding vertices and edges through practical exercises. The supplementary activity further reinforced our understanding by relating DFS to the real world, such as when we first explore the island, paralleling DFS with preorder tree traversal. Putting it all together, these exercises filled in our understandings of graph structures and traversal methods.

In summary, this activity proved to be very valuable in solidifying our knowledge about graphs, their data structures, and their applications. The combination of theoretical insights and practical exercises gave us a very effective learning experience. We are therefore now in a position to feel more confident in the implementation of graph algorithms for direct practice. Our objectives would therefore include some advanced knowledge of the sophisticated graph algorithms and various available graph libraries. This way we could expand our expertise and implement more complex graph implementations.