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): Bona, Andrei Nycole So. Masangkay, Frederick D. Roallos, Jean Gabriel Vincent G. Santos, Andrei R. Zolina, Anna Marie L.	**Instructor:** Professor Maria Rizette Sayo

A. Output(s) and Observation(s)

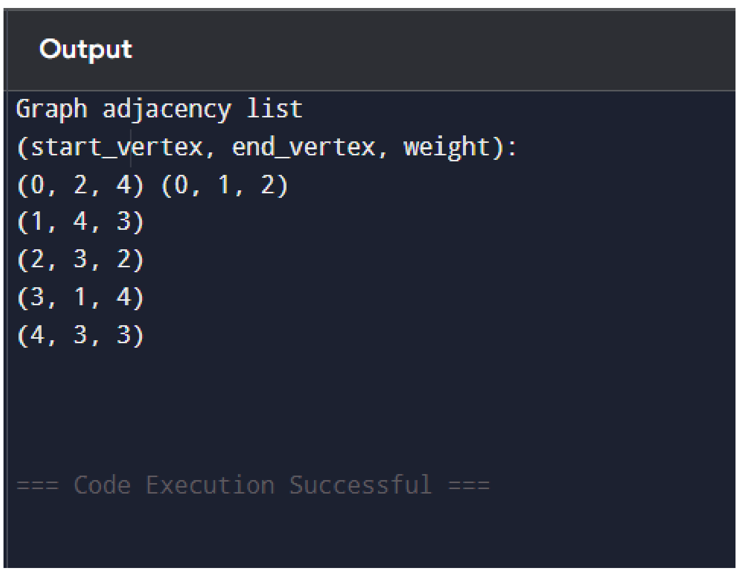
ILO A: Create C++ code for graph implementation utilizing adjacency matrix and adjacency list

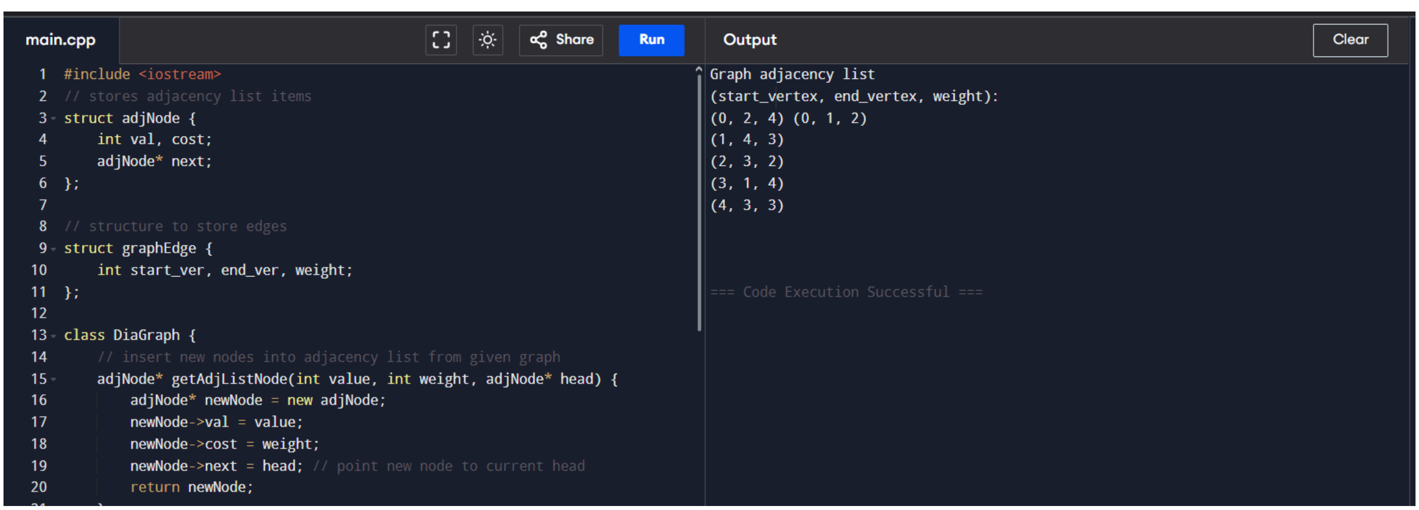
```
#include <iostream>
// stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};

// 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 != nullptr) {
                adjNode* temp = current;
                current = current->next;
                delete temp;
            }
        }
        delete[] head; // delete array of pointers
    }
};

// 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:





ILO B: Create C++ code for implementing graph traversal algorithms such as Breadth-First and Depth-FirstSearch

B.1. Depth-First Search

```
#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;
    }
};

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:
```

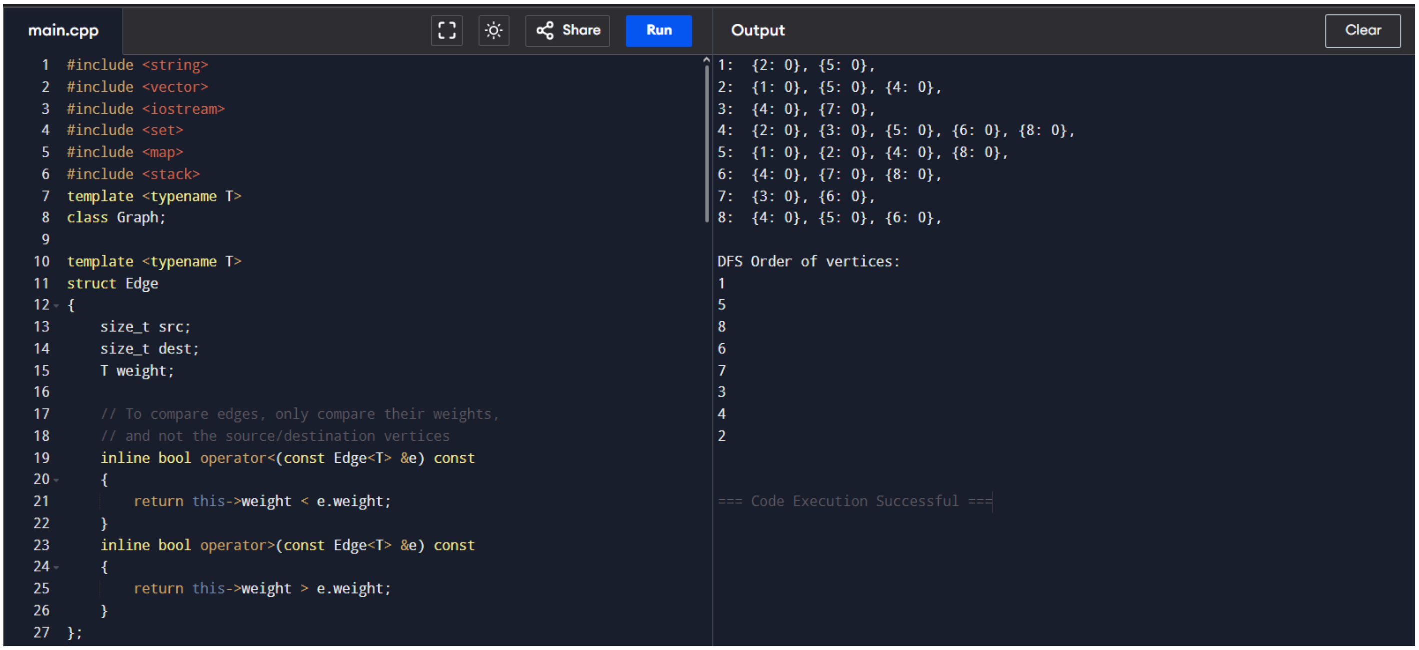
```
    // Initialize the graph with N vertices
    Graph(size_t N) : V(N) {}

    // 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 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 dest)
{
    std::stack<size_t> stack;
```

```
    std::vector<size_t> visit_order;
    std::set<size_t> visited;

    stack.push(1); // Assume that DFS always starts from vertex ID 1

    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())
                {
                    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;
}
```

OUTPUT:

```
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
```




```
#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;
    }
};
```

```
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


=== Code Execution Successful ===
```

B.2. Breadth-First Search

```
#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++) {
        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:
    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;
};

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, 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;
}

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

    queue.push(1); // Assume that 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);
            for (auto e : G.outgoing_edges(current_vertex)) {
                queue.push(e.dest);
            }
        }
    }
    return visit_order;
}

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

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


=== Code Execution Successful ===
```

main.cpp   Share   Run

```
#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;
    }
};
```

Output   Clear

```
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


=== Code Execution Successful ===
```

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 another vertex from the same vertex. Discuss which algorithm would be most helpful to accomplish this task.**

- We believe that Depth First Search (DFS) is the best algorithm since it enables us to fully explore one path before going back and then exploring other paths from the same vertex. This approach thoroughly explores and retraces its steps, making it effective for visiting every location.

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.**

- Depending on the order in which the nodes are visited, we consider that pre-order, in-order, or post-order traversal is the equivalent of DFS in tree traversal. Before proceeding to the next subtree, these techniques visit a node and its descendants in accordance with the DFS approach.

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

- The data structure that was used in Breadth First Search (BFS) is a queue. The queue makes sure that the nodes are processed level by level, maintaining the correct order of exploration.

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

- We can say that every node in BFS can only be visited once. This avoids cycles and redundant operations by guaranteeing that a node is not revisited after it has been processed.

C. Conclusion & Lessons Learned

The algorithms Depth First Search (DFS) and Breadth First Search (BFS) are critical for graph exploration, we concluded. DFS explores a graph in depth, whereas BFS moves through nodes level by level. We learned the value of employing data structures such as queues for BFS and stacks for DFS through the use of these algorithms. Our learning of the algorithms' behavior was enhanced by the ILOs. For example, these algorithms can be applied to way or route planning. Our graphing abilities have improved as a result of finishing this task.

D. Assessment Rubric

E. External References