Activity No. 10	
GRAPHS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/13/2024
Section: CPE21S1	**Date Submitted:** 11/15/2024
Name(s): Abarabar, John Nathan R. Aduana, Jake Norwin Bulambao, Adrian Justin M. Gaspar, Aaron Rowen Potestades, North Nygel G.	**Instructor:** Engr. Sayo

6. Output

ILO A

A.1

```
    // Add a vertex to the graph
    void addVertex() {
        // Allocate a new array of pointers to represent the adjacency list
        adjNode **newHead = new adjNode*[N + 1];

        // Copy existing vertices' adjacency lists to the new array
        for (int i = 0; i < N; i++) {
            newHead[i] = head[i];
        }

        // Initialize the new vertex with no edges (null pointer)
        newHead[N] = nullptr;

        // Delete the old adjacency list array and update head to the new one
        delete[] head;
        head = newHead;
        N++;  // Increase the number of vertices
        std::cout << "Vertex " << N - 1 << " added.\n";
    }

    // Add an edge between two vertices
    void addEdge(int start, int end, int weight) {
        // Ensure the edge is added to the adjacency list of the start vertex
        adjNode* newNode = getAdjListNode(end, weight, head[start]);
        head[start] = newNode;
        std::cout << "Edge added from vertex " << start << " to vertex " << end << " with weight " << weight << ".\n";
    }

    // Display the graph vertices
    void displayVertices() {
        std::cout << "Graph vertices: ";
        for (int i = 0; i < N; i++) {
            std::cout << i << " ";
        }
        std::cout << std::endl;
    }
};
```

A.2

```
// Function to create the adjacency matrix from the adjacency list
void createAdjMatrix() {
    // Create the adjacency matrix initialized to 0 (no edge)
    int **adjMatrix = new int*[N];
    for (int i = 0; i < N; i++) {
        adjMatrix[i] = new int[N]();
        for (int j = 0; j < N; j++) {
            adjMatrix[i][j] = 0;  // No edge, initialized to 0
        }
    }

    // Populate the adjacency matrix from the adjacency list
    for (int i = 0; i < N; i++) {
        adjNode* temp = head[i];
        while (temp != nullptr) {
            adjMatrix[i][temp->val] = temp->cost;  // Set weight
            temp = temp->next;
        }
    }

    // Display the adjacency matrix
    std::cout << "\nAdjacency Matrix Representation (Weight):\n";
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            std::cout << adjMatrix[i][j] << " ";
        }
        std::cout << std::endl;
    }

    // Clean up the memory used by the adjacency matrix
    for (int i = 0; i < N; i++) {
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;
}
```

A.3

```
// 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;
}
```

ILO B

B.1

```
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

```
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
```

7. 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 algorithm that would be most helpful to accommodate this task would be the Depth First Search, as the flow of the person's needs is the exact same flow that a DFS is performed. They travel from the first vertex and visit every vertex possible in one path, and then it backtracks to the first vertex with another path to explore. It repeats this process until each vertex is labeled as visited, following the same flow as how the person wants to visit different locations.

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.

- The equivalent of DFS for a tree would be a post-order traversal, as they both start from the bottom and make their way up their respective data structures.

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

- The data structure used in the implementation of the Breadth First Search is a queue, which is the opposite of the Depth First Search, which uses a stack implementation.

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

- A node can only be visited once in a BFS, as the node is marked visited once the BFS reaches that node.

8. Conclusion

In conclusion, graphs are a complex utilization of the lessons we have learned up to this day, such as queues and stacks intermingling with arrays and pointers from linked lists. The procedure was easy enough, as the second section was pretty much copy-and-paste to get the output, and the first part was harder as we had to code it ourselves with a little help from the document. The supplementary activity was the most difficult, with the coding needed in number 2 especially, though the rest were relatively easy. In the future we could improve in terms of being faster at finishing the code and arranging the document, but we did well for our output.

9. Assessment Rubric