Hands-on Activity 10.1

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): Avila, Vince Gabriel V.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILOA.A

Adjacency List:

- Edges are organized to construct a graph and maintain all neighboring vertices for each vertex with linked lists. This data structure allows for a compact representation of the graph edges and efficient traversal of the edges. After constructing the graph, the program prints all edges of the graph displaying the originating vertex, terminating vertex and weight value associated with the edge. Through this process, the program provides a clear view of the connection between vertices and associated cost for each edge.

```
public:
    ListNode** adjacencyArray;

    DirectedGraph(EdgeData edgeList[], int edgeCount, int vertices) {
        vertexCount = vertices;
        adjacencyArray = new ListNode*[vertexCount]();

        for (int i = 0; i < vertexCount; ++i) {
            adjacencyArray[i] = nullptr;
        }

        for (int i = 0; i < edgeCount; ++i) {
            int start = edgeList[i].srcNode;
            int end = edgeList[i].destNode;
            int weight = edgeList[i].weightVal;

            ListNode* node = createListNode(end, weight, adjacencyArray[start]);
            adjacencyArray[start] = node;
        }
    }

    ~DirectedGraph() {
        for (int i = 0; i < vertexCount; ++i) {
            ListNode* current = adjacencyArray[i];
            while (current != nullptr) {
                ListNode* toDelete = current;
                current = current->nextNode;
                delete toDelete;
            }
        }
        delete[] adjacencyArray;
    }
};

// Function to print adjacency list of a given vertex
void printAdjacencyList(ListNode* listHead, int vertex) {
    while (listHead != nullptr) {
        std::cout << "(" << vertex << ", " << listHead->dest
                  << ", " << listHead->edgeWeight << ") ";
```

```cpp

#include <iostream>

// Node structure for adjacency list
struct ListNode {
    int dest;
    int edgeWeight;
    ListNode* nextNode;
};

// Edge structure for graph edges
struct EdgeData {
    int srcNode;
    int destNode;
    int weightVal;
};

class DirectedGraph {
    int vertexCount;

    // Helper to create a new adjacency list node
    ListNode* createListNode(int destination, int w, ListNode* head) {
        ListNode* newNode = new ListNode;
        newNode->dest = destination;
        newNode->edgeWeight = w;
        newNode->nextNode = head;
        return newNode;
    }

public:
    ListNode** adjacencyArray;

    DirectedGraph(EdgeData edgeList[], int edgeCount, int vertices) {
        vertexCount = vertices;
        adjacencyArray = new ListNode*[vertexCount]();

        for (int i = 0; i < vertexCount; ++i) {
            adjacencyArray[i] = nullptr;
```

```
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)


--------------------------------
Process exited after 0.3245 seconds with return value 0
Press any key to continue . . .
```

ILO.B.1

Adjacency list implementation:

- This program constructs a graph with weighted connections and traverses it using depth-first search. It creates a graph with 8 points and stores the connections in a list of edges. The depth-first search uses a stack to keep track of which vertices to visit next and makes sure each vertex is only visited once, so it is not revisited. Then, it outputs the list of neighbors for each vertex and the order of vertices the program visited during the depth-first search.

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <stack>

template <typename WeightType>
class Network;

template <typename WeightType>
struct Connection {
    size_t origin;
    size_t target;
    WeightType cost;

    bool operator<(const Connection<WeightType>& other) const {
        return cost < other.cost;
    }
    bool operator>(const Connection<WeightType>& other) const {
        return cost > other.cost;
    }
};

template <typename WeightType>
class Network {
public:
    explicit Network(size_t totalNodes) : nodesCount(totalNodes) {}

    size_t node_count() const { return nodesCount; }

    const std::vector<Connection<WeightType>>& get_connections() const { return connectionsList; }

    void add_connection(const Connection<WeightType>& connection) {
        if (connection.origin >= 1 && connection.origin <= nodesCount &&
            connection.target >= 1 && connection.target <= nodesCount) {
            connectionsList.push_back(connection);
        } else {
            std::cerr << "Error: Node index out of valid range" << std::endl;
        }
```

```cpp
            connection.target >= 1 && connection.target <= nodesCount) {
            connectionsList.push_back(connection);
        } else {
            std::cerr << "Error: Node index out of valid range" << std::endl;
        }
    }

    std::vector<Connection<WeightType>> connections_from_node(size_t node) const {
        std::vector<Connection<WeightType>> results;
        for (const auto& c : connectionsList) {
            if (c.origin == node) {
                results.push_back(c);
            }
        }
        return results;
    }

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

private:
    size_t nodesCount;
    std::vector<Connection<WeightType>> connectionsList;
};

template <typename WeightType>
std::ostream& operator<<(std::ostream& os, const Network<WeightType>& net) {
    for (size_t nodeId = 1; nodeId <= net.node_count(); ++nodeId) {
        os << nodeId << ": ";
        auto outgoing = net.connections_from_node(nodeId);
        for (const auto& conn : outgoing) {
            os << "{" << conn.target << ": " << conn.cost << "}, ";
        }
        os << "\n";
    }
```

```
            connection.target >= 1 && connection.target <= nodesCount) {
            connectionsList.push_back(connection);
        } else {
            std::cerr << "Error: Node index out of valid range" << std::endl;
        }
    }

    std::vector<Connection<WeightType>> connections_from_node(size_t node) const {
        std::vector<Connection<WeightType>> results;
        for (const auto& c : connectionsList) {
            if (c.origin == node) {
                results.push_back(c);
            }
        }
        return results;
    }

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

private:
    size_t nodesCount;
    std::vector<Connection<WeightType>> connectionsList;
};

template <typename WeightType>
std::ostream& operator<<(std::ostream& os, const Network<WeightType>& net) {
    for (size_t nodeId = 1; nodeId <= net.node_count(); ++nodeId) {
        os << nodeId << ": ";
        auto outgoing = net.connections_from_node(nodeId);
        for (const auto& conn : outgoing) {
            os << "{" << conn.target << ": " << conn.cost << "}, ";
        }
        os << "\n";
    }
```

```
}

template <typename WeightType>
std::vector<size_t> perform_dfs(const Network<WeightType>& net, size_t startNode) {
    std::stack<size_t> nodeStack;
    std::set<size_t> visitedNodes;
    std::vector<size_t> visitSequence;

    nodeStack.push(startNode);

    while (!nodeStack.empty()) {
        size_t currentNode = nodeStack.top();
        nodeStack.pop();

        if (visitedNodes.find(currentNode) == visitedNodes.end()) {
            visitedNodes.insert(currentNode);
            visitSequence.push_back(currentNode);

            auto neighbors = net.connections_from_node(currentNode);
            for (const auto& c : neighbors) {
                if (visitedNodes.find(c.target) == visitedNodes.end()) {
                    nodeStack.push(c.target);
                }
            }
        }
    }
    return visitSequence;
}

template <typename WeightType>
Network<WeightType> generate_sample_network() {
    Network<WeightType> net(8);

    std::map<size_t, std::vector<std::pair<size_t, WeightType>>> adjacencyMap = {
        {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}}},
```

```
    Network<WeightType> net(8);

    std::map<size_t, std::vector<std::pair<size_t, WeightType>>> adjacencyMap = {
        {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}}}
    };

    for (const auto& [originNode, neighbors] : adjacencyMap) {
        for (const auto& [targetNode, weightVal] : neighbors) {
            net.add_connection(Connection<WeightType>{originNode, targetNode, weightVal});
        }
    }
    return net;
}

template <typename WeightType>
void test_dfs_traversal() {
    auto network = generate_sample_network<WeightType>();

    std::cout << "Network adjacency list:\n" << network << std::endl;

    std::cout << "DFS traversal order of nodes:\n";
    auto traversalOrder = perform_dfs(network, 1);
    for (auto node : traversalOrder) {
        std::cout << node << "\n";
    }
}

int main() {
    using Weight = unsigned;
    test_dfs_traversal<Weight>();
    return 0;
```

```
Network adjacency list:
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 traversal order of nodes:
1
5
8
6
7
3
4
2


--------------------------------
Process exited after 0.3528 seconds with return value 0
Press any key to continue
```

ILO.B.2

- The following code creates a graph in which each connection has a weight, and subsequently explores the graph using BFS. The graph is created with eight nodes, and the edges and their weights are stored in a list. BFS visits each of the nodes one at a time using a queue, starting with node 1 and traversing the graph level by level. At the end of the program execution, it will output the nodes connected and the in which they were visited during the search.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <map>

// Edge structure representing a connection between nodes
struct Link {
    int startNode;
    int endNode;
    unsigned cost;
};

// Simple Graph class
class Network {
public:
    Network(int totalNodes) {
        this->totalNodes = totalNodes;
    }

    void addLink(int fromNode, int toNode, unsigned weight) {
        if (fromNode >= 1 && fromNode <= totalNodes && toNode >= 1 && toNode <= totalNodes) {
            connections.push_back({fromNode, toNode, weight});
        } else {
            std::cout << "Error: node index is out of bounds\n";
        }
    }

    std::vector<Link> linksFrom(int node) const {
        std::vector<Link> outgoing;
        for (const auto& link : connections) {
            if (link.startNode == node) {
                outgoing.push_back(link);
            }
        }
        return outgoing;
    }

    int nodeCount() const {
```

```
        return totalNodes;
    }

    void showNetwork() const {
        for (int node = 1; node <= totalNodes; ++node) {
            std::cout << node << ": ";
            auto neighbors = linksFrom(node);
            for (const auto& l : neighbors) {
                std::cout << "(" << l.endNode << ", " << l.cost << ") ";
            }
            std::cout << std::endl;
        }
    }

private:
    int totalNodes;
    std::vector<Link> connections;
};

// Create example network graph
Network buildSampleNetwork() {
    Network net(8);

    std::map<int, std::vector<std::pair<int, unsigned>>> nodeLinks = {
        {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& [source, destinations] : nodeLinks) {
        for (const auto& [target, cost] : destinations) {
            net.addLink(source, target, cost);
        }
    }

    return net;
}

// Breadth First Search traversal
std::vector<int> breadthFirstTraversal(const Network& net, int start) {
    std::queue<int> visitQueue;
    std::set<int> visitedNodes;
    std::vector<int> orderVisited;

    visitQueue.push(start);

    while (!visitQueue.empty()) {
        int currentNode = visitQueue.front();
        visitQueue.pop();

        if (visitedNodes.find(currentNode) == visitedNodes.end()) {
            visitedNodes.insert(currentNode);
            orderVisited.push_back(currentNode);

            auto adjacent = net.linksFrom(currentNode);
            for (const auto& link : adjacent) {
                if (visitedNodes.find(link.endNode) == visitedNodes.end()) {
                    visitQueue.push(link.endNode);
                }
            }
        }
    }

    return orderVisited;
}

int main() {
    Network myNetwork = buildSampleNetwork();

    std::cout << "Network adjacency list:\n";
    myNetwork.showNetwork();

    std::cout << "\nBFS traversal starting from node 1:\n";
```

```
Network 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 starting from node 1:
1
2
5
4
8
3
6
7


--------------------------------
Process exited after 0.3463 seconds with return value 0
Press any key to continue . . .
```

B. Answers to Supplementary Activity

ILO C: Demonstrate an understanding of graph implementation, operations and traversal methods.

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 most suitable algorithm for traversing all locations starting from one place and backtracking when necessary is Depth First Search (DFS). DFS will take a step down one path until it can't go any further, and then backtrack to find a different path. It's similar to investigating one road completely before checking others. This is beneficial to a person who wants to make every event in order. DFS makes use of a stack or recursion to keep track of where a person should backtrack to.

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.**

   DFS in trees is akin to preorder traversal. In preorder, we start at the root, go down the left side, and then go down the right side. Both DFS and preorder go deep before they go wide. They will completely explore one branch before moving on to another one. You can accomplish this in code with recursion or with a stack.

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

   BFS Breadth First Search works using a queue. A queue visits nodes level by level, starting at the first node. When a node is visited, the neighbors of that node are added to the queue so that we can be sure to visit the

closest nodes first. It keeps nodes in the order in which they were added. The queue ensures that we visit all nodes close to the first node before going deeper. This follows the First In, First Out FIFO rule, meaning that the node which was added first is the next one to be visited.

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

In BFS, a node can be visited just once. That way the algorithm doesn't go in a circle or duplicate work. When a node is visited, it's marked so as not to visit it again this ensures BFS is both efficient and correct. Without marking, it would be possible to visit the same node many times.

C. Conclusion & Lessons Learned

Provide the following:

1. **Summary of lessons learned**
   - In this assignment, I learned how to use adjacency lists to represent how a graph is connected. I also learned about search algorithms like breadth first search BFS. BFS essentially uses a queue to search the graph level by level. I also noticed that depth-first search DFS does the opposite; it goes as far down a graph along a path before returning to find an alternative option. Both of these methods are helpful to solve problems like the shortest path or to find a path that is not connected in a network.

2. **Analysis of the procedure**
   - The graph was created by connecting nodes with weighted edges, and traversal using a queue and visited set. The BFS traversal worked in logical steps, and the order in which I visited nodes was what I expected. The visited set prevented the algorithm from doubling back on previously visited nodes, which ultimately made the constraint faster and more efficient.

3. **Analysis of the supplementary activity**
   - This additional exercise really helped me visualize the differences between BFS and DFS and when to apply each. BFS is good for either finding the shortest path or for going level by level in the search, while DFS is better for problems that require backtracking or deep digging. Examining both the pseudocode and C++ helped to conceptualize the ideas and understand how they work in real programs.

4. **Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?**
   - This activity helped me understand how BFS works, and that it is similar to tree traversal. I still need to practice writing better pseudocode, and drawing clearer graphs of the traversal. I learned where I need improvement, and this activity increased my confidence with algorithms.

D. Assessment Rubric

E. External References