ACTIVITY NO. 10

GRAPHS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 09/30/25**
Section: CPE21S4	**Date Submitted: 09/30/25**
Name: Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

1. Objective(s)

Create C++ code for implementation of Graph ADT.

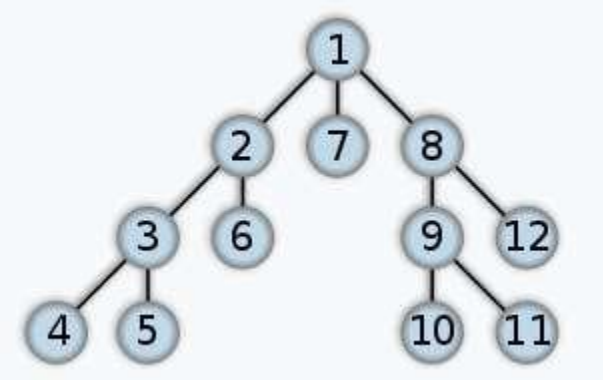
2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Create C++ code for graph implementation utilizing adjacency matrix and adjacency list.
- Create C++ code for implementing graph traversal algorithms such as Breadth-First and Depth-First Search.
- Demonstrate an understanding of graph implementation, operations and traversal methods.

3. Discussion

For a given vertex v, the depth-first search algorithm proceeds along a path from v as deeply into the graph as possible before backing up. That is, after visiting a vertex v, the depth-first search algorithm visits (if possible) an unvisited adjacent vertex to vertex v. The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v. We may visit the vertices adjacent to v in sorted order.



The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

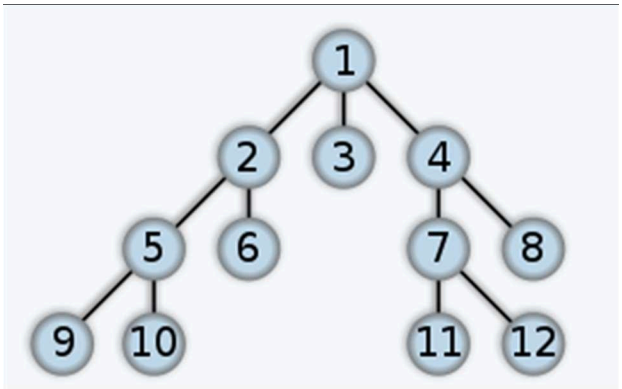
Recursive Depth-First Search Algorithm

```
dfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
      Mark v as visited;
      for (each unvisited vertex u adjacent to v)
            dfs(u)
}
```

Iterative Depth-First Search Algorithm

```
dfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
      s.createStack();
      // push v into the stack and mark it
      s.push(v);
      Mark v as visited;
      while (!s.isEmpty()) {
         if (no unvisited vertices are adjacent to the vertex on
             the top of stack)
            s.pop(); // backtrack
         else {
            Select an unvisited vertex u adjacent to the vertex
               on the top of the stack;
            s.push(u);
            Mark u as visited;
         }
      }
}
```

After visiting a given vertex v, the breadth-first search algorithm visits every vertex adjacent to v that it can before visiting any other vertex. The breadth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v. However, we may visit the vertices adjacent to v in sorted order.



There are many ways to traverse graphs. BFS is the most commonly used approach. BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Iterative Breadth-First Search Algorithm

```
bfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using breath-first strategy: Iterative Version
      q.createQueue();
      // add v to the queue and mark it
      q.enqueue(v);
      Mark v as visited;
      while (!q.isEmpty()) {
         q.dequeue(w);
         for (each unvisited vertex u adjacent to w) {
            Mark u as visited;
            q.enqueue(u);
         }
      }
}
```

4. Materials and Equipment

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

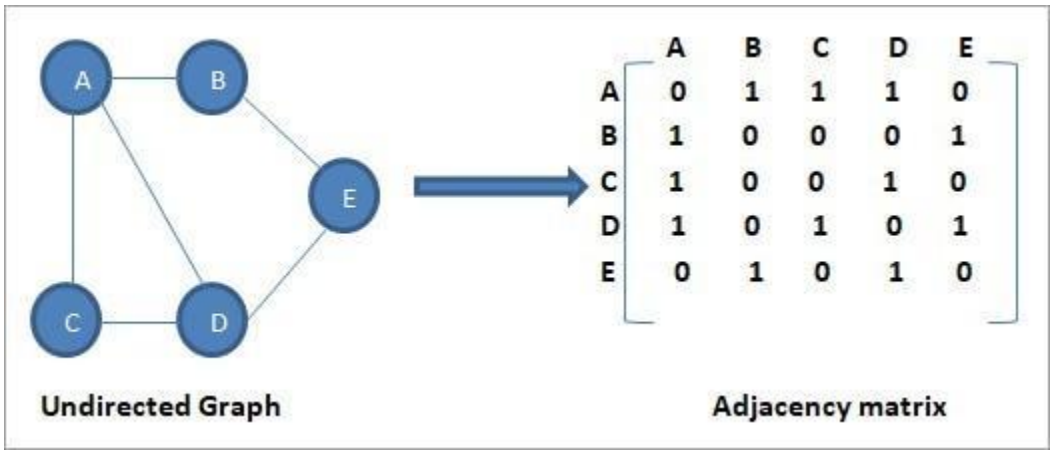
ILO A: Create C++ code for graph implementation utilizing adjacency matrix and adjacency list

A.1. Create a Graph

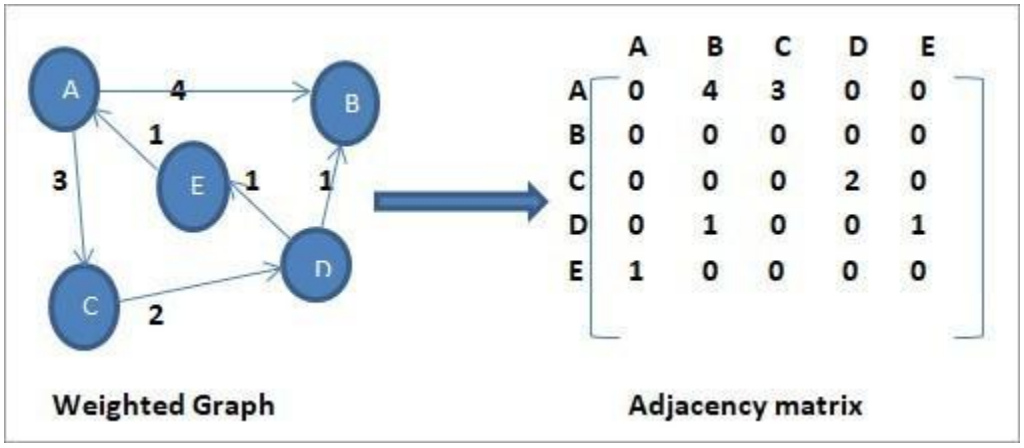
Following are the basic operations that we can perform on the graph data structure:

- **Add a vertex:** Adds vertex to the graph.
- **Add an edge:** Adds an edge between the two vertices of a graph.
- **Display the graph vertices:** Display the vertices of a graph.

A.2. Adjacency Matrix

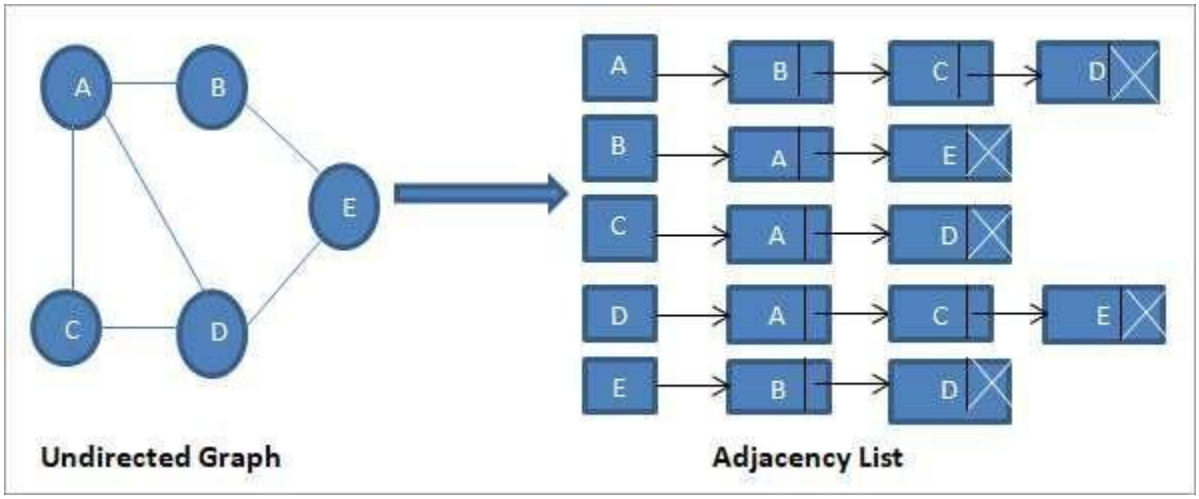


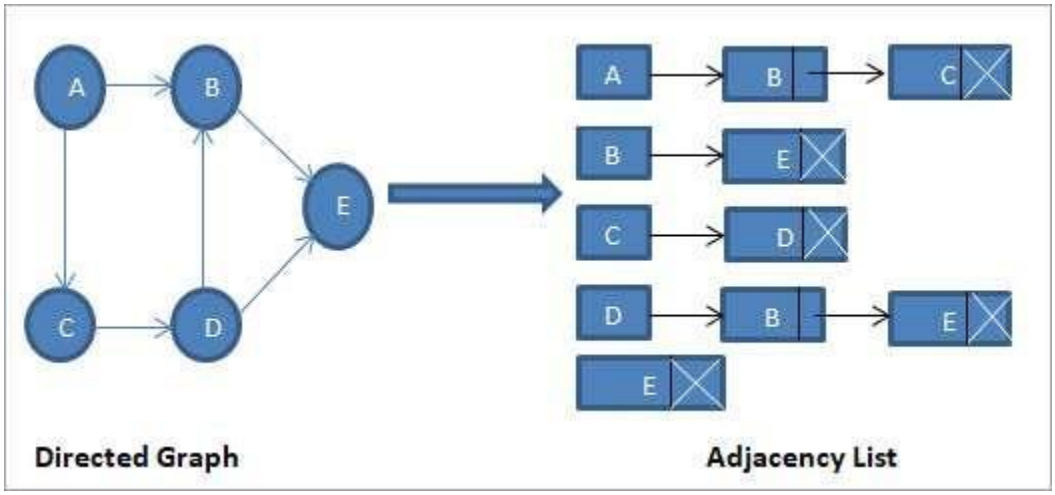
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0



	A	B	C	D	E
A	0	4	3	0	0
B	0	0	0	0	0
C	0	0	0	2	0
D	0	1	0	0	1
E	1	0	0	0	0

A.3. Adjacency List





Sample 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++)
                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;
    }
```

Implement the given code and indicate your output as a table in section 6.

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

B.1. Depth-First Search

Step 1. Include the required header files, as follows:

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

template <typename T>
class Graph;
```

Step 2. Write the following struct in order to implement an edge in our 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;
    }
};
```

Step 3. Next, overload the << operator for the graph so that it can be printed out using the following function:

```
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;
    }
```

Step 4. Implement the graph data structure that uses an edge list representation as follows:

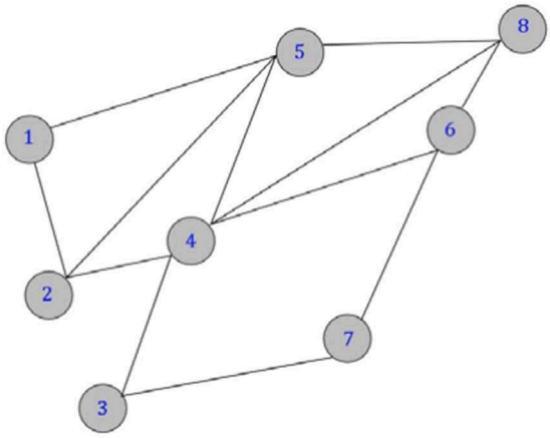
```
    template <typename T>
    class Graph
    {
    public:
        // Initialize the graph with N vertices
        Graph(size_t N) : V(N)
        {
        }
        // Return number of vertices in the graph
        auto vertices() const
        {
            return V;
        }
        // Return 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;
};
```

Step 5. Now, we need a function to perform DFS for our graph. Implement it as follows:

```
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;
}
```

Step 6. We shall test our implementation of the DFS on the graph shown here:



Use the following function to create and return the graph:

```
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;
}
```

Note the use of null values for edge weights since DFS does not require edge weights. A simpler implementation of the graph could have omitted the edge weights entirely without affecting the behavior of our DFS algorithm.

Step 7. Finally, add the following test and driver code, which runs our DFS implementation and prints the output:

```
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;
}
```

Step 8. Compile and run the preceding code. **Include your output as a table in section 6.**

B.2. Breadth-First Search

Step 1: Include the required header files and declare the graph as follows:

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

template <typename T>
class Graph;
```

Step 2: Write the following struct, which represents an edge in our 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;
    }
};
```

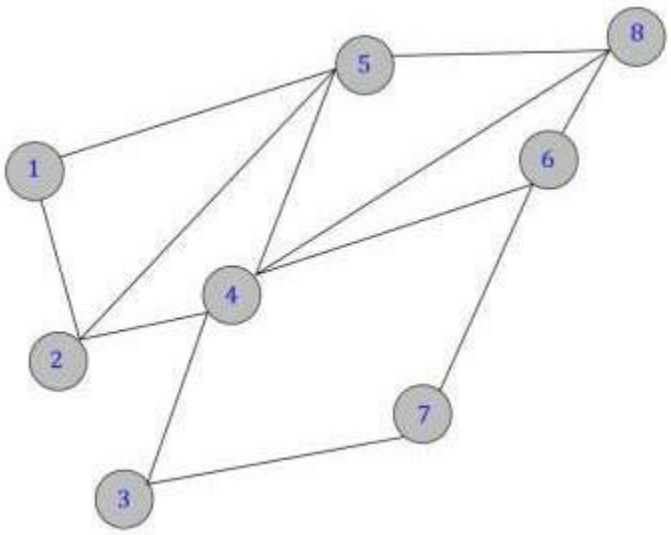
Step 3: Next, overload the << operator for the Graph data type in order to display the contents of the graph:

```
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;
    }
}
```

Step 4: Write a class to define our graph data structure, as shown here:

```
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 T>
    friend std::ostream &operator<<(std::ostream &os, const Graph<T> &G);

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

Step 5: For this exercise, we shall test our implementation of BFS on the following graph:



We need a function to create and return the required graph. Note that while edge weights are assigned to each edge in the graph, this is not necessary since the BFS algorithm does not need to use edge weights. Implement the function as follows:

```
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;
}
```

Step 6: Implement the breadth-first search like so:

```
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;
}
```

Step 7: Add the following test and driver code that creates the reference graph, runs BFS starting from vertex 1, and outputs the results:

```
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;
}
```

Include the out and observation as a table in section 6.

6. Output

ILO.A

Adjacency list

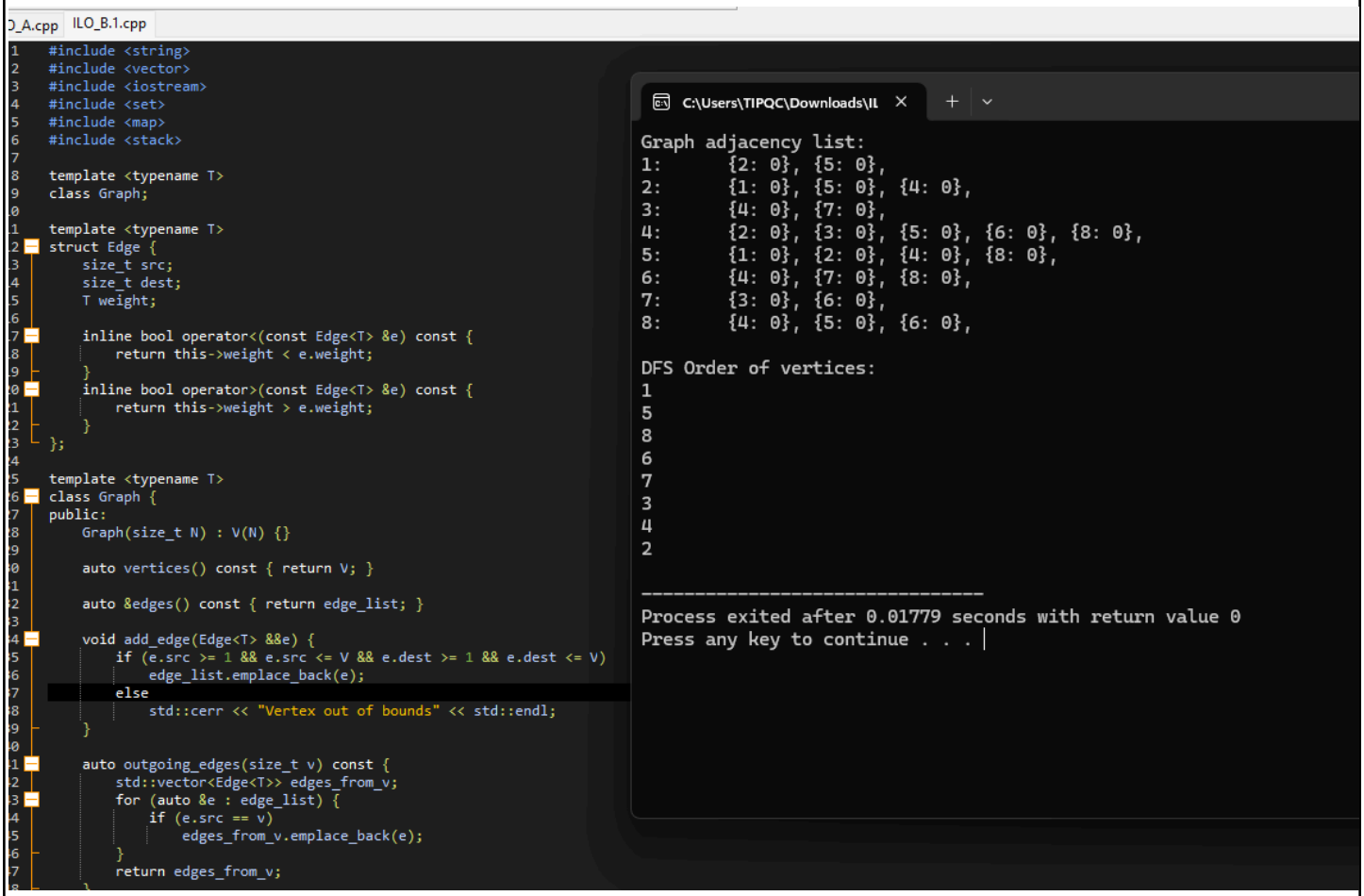
- This code builds the graph from a list of edges and stores each vertex's neighbors in linked lists. The program then prints each edge in the form start, end and weight. It clearly shows how nodes are connected and their respective edge costs.

```
            current = current->next;
            delete temp;
        }
    }
    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.
    // (x, y, w) -> edge from x to y with weight w
    graphEdge edges[] = {
        {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):"

    for (int i = 0; i < N; i++) {
        // display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }

    return 0;
}
```

```
C:\Users\TIPQC\Downloads\IL
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 0.01061 seconds with return value 0
Press any key to continue . . .
```

ILO.B.1
Adjacency list implementation

- This code shows a graph that supports weighted edges and uses a depth-first search traversal. It creates a reference graph with eight vertices and connections stored in a vector of edges. The DFS is implemented using a stack to track the order of exploration, ensuring each vertex is visited only once. Finally, the program prints the adjacency list and displays the order in which vertices are visited during DFS.



ILO.B.2

- This code is also an adjacency list implementation of a weighted graph that performs breadth-first search traversal. It constructs a reference graph with eight vertices and edges that include weights stored in a vector. But this uses BFS as a queue to systematically visit each vertex starting from vertex 1, ensuring that nodes are processed in level order. In the end, the program prints both the adjacency list representation of the graph and the sequence of vertices visited during BFS.

ILO_A.cpp | ILO_B.1.cpp | ILO_B.2.cpp

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

// Forward declaration
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>
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;
```

```
C:\Users\TIPQC\Downloads\IL
Graph 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 Order of vertices:
1
2
5
4
8
3
6
7

--------------------------------
Process exited after 0.02417 seconds with return value 0
Press any key to continue . . .
```

7. 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 best algorithm for the person who wants to explore as far as possible along one path, then backtrack and continue with other paths, is Depth First Search (DFS). In tree traversal strategies, DFS is equivalent to preorder, inorder, or postorder traversals, depending on the order of visiting nodes.

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 behaves like following one narrow path until no more moves are possible, then backtracking to try other branches. While BFS, by contrast, explores all immediate neighbors first before moving deeper.

3. In the performed code, what data structure is used to implement the Breadth First Search?
    - In the provided BFS code, traversal uses `std::queue<size_t>` for ordering and a `std::set<size_t>` to track visited nodes.

4. How many times can a node be visited in the BFS?
    - The BFS implementation uses a queue to manage traversal order and ensures nodes are not revisited by checking the visited set. As a result, each node is visited exactly once during BFS traversal.

8. Conclusion

Provide the following:

- Summary of lessons learned
    - From this activity, I learned how to represent a graph using adjacency lists and implement traversal strategies such as breadth-first search. I understood how BFS works by visiting vertices level by level and how the use of a queue ensures systematic exploration. I also saw how DFS differs in behavior by going deeper into a path before backtracking.

- Analysis of the procedure
    - The procedure shows how the graph was constructed with weighted edges and how traversal was managed with a queue and a visited set. The step-by-step BFS traversal output matched the expected order of visiting vertices. It confirmed that every node is only visited once due to the visited set, making the algorithm efficient.

- Analysis of the supplementary activity
    - The supplementary task helped me see the difference between BFS and DFS and gave me a deeper appreciation of when each algorithm is most useful. BFS is good for shortest path or level-order tasks, while DFS fits scenarios where backtracking and full exploration are needed. Looking at both pseudocode and the actual C++ implementation solidified the connection between theory and practice.

- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
    - It helps me a lot through understanding the traversal process, and relating BFS to tree traversal strategies. However, I could improve in writing cleaner pseudocode on my own and strengthening my ability to draw graphical representations of traversal. This activity gave me more confidence in analyzing algorithms but also showed me areas where I can practice more.

9. Assessment Rubric