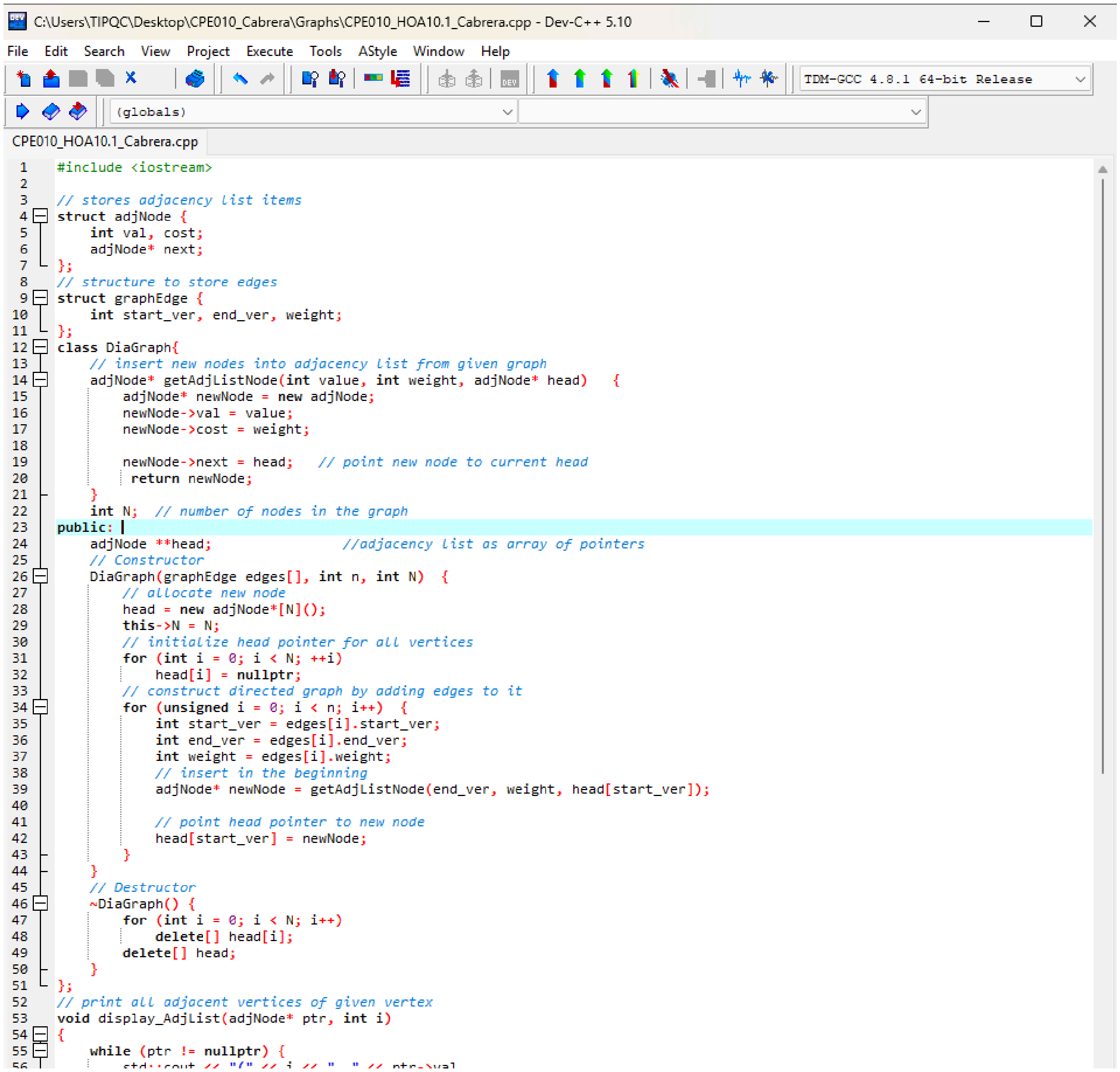
Hands-on Activity 10.1	
Graphs	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09/30/2025
Section: CPE21S4	**Date Submitted:** 09/30/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado
6. Output	

ILO A

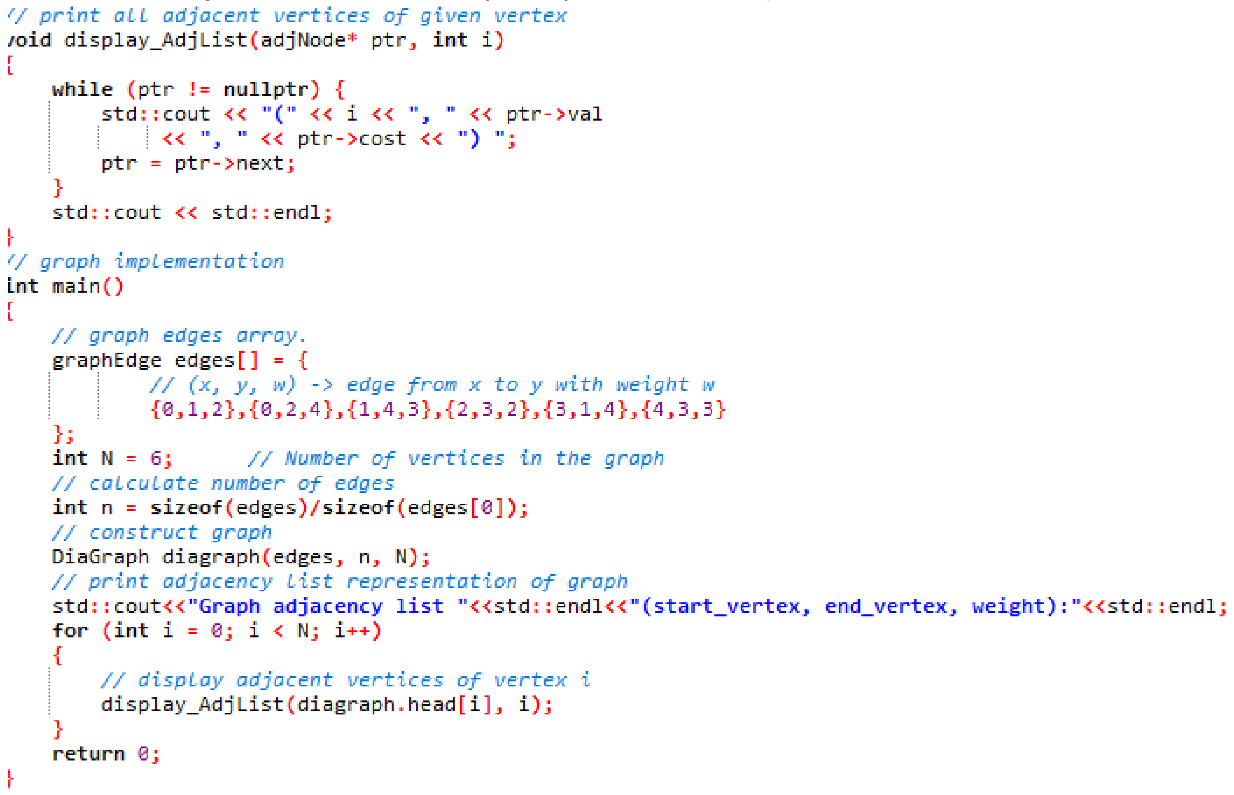

C:\Users\TIPQC\Desktop\CPE010_Cabrera\Graphs\CPE010_HOA10.1_Cabrera.cpp - Dev-C++ 5.10
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.8.1 64-bit Release
(globals)
CPE010_HOA10.1_Cabrera.cpp

```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
```

The program starts by declaring a structure named adjNode, in it are the int variables val and cost and the reference pointer next to be used as the node pointers for the DiaGraph class.

Another structure declared is the graphEdge to include which vertex is the start or the end, it also includes the weight of the edge to represent cost.

The DiaGraph class starts by declaring private pointers for adjNode, value, and weight, this is so that the user can't adjust these values by themselves and so a new Node gets their values correctly whenever they are added. A constructor is made for the heads of the graph, where the adjacency list would start, the DiaGraph class will also be called by itself to initiate the making of each value in the adjacency list. The DiaGraph class ends with a destructor function.

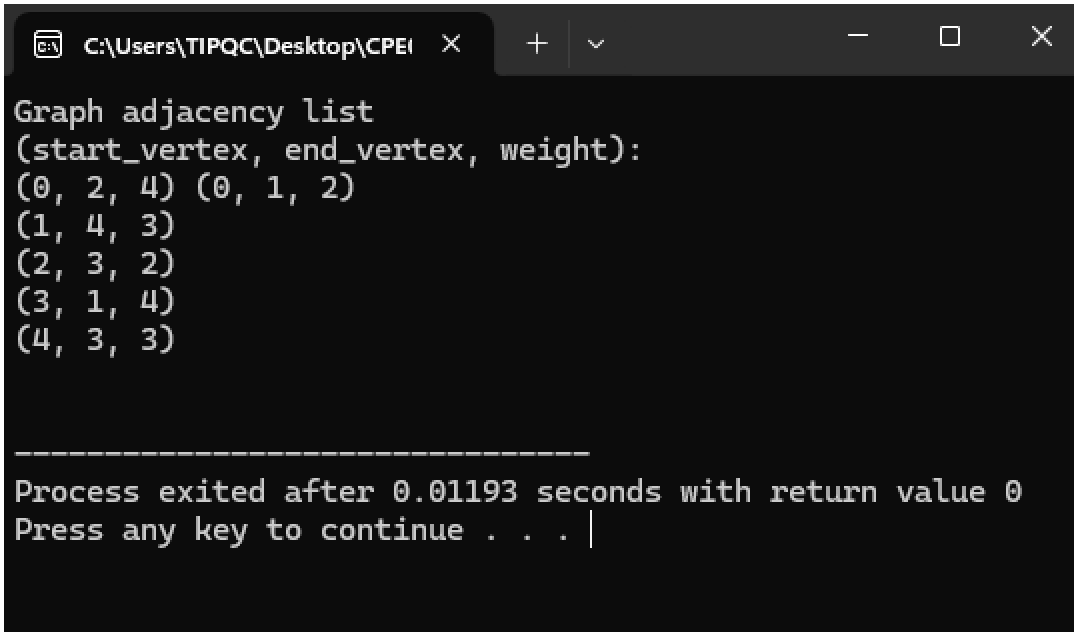


```
// print all adjacent vertices of given vertex
/oid display_AdjList(adjNode* ptr, int i)
{
    while (ptr != nullptr) {
        std::cout << "(" << i << ", " << ptr->val
            << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    std::cout << std::endl;
}
// graph implementation
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;
}
```

For the display_AdjList function, the code is similar to displaying linked lists, where the code runs through the ptr references using a while loop until the ptr is equal to the nullptr, which is the end of the graph, each loop displays the start vertex, end vertex, and cost of the edge connecting them.

The main function just calls the graphEdge function to make an array of adjacent vertices and then display them using the display_AdjList function.

ILO A - Graph Implementation Code



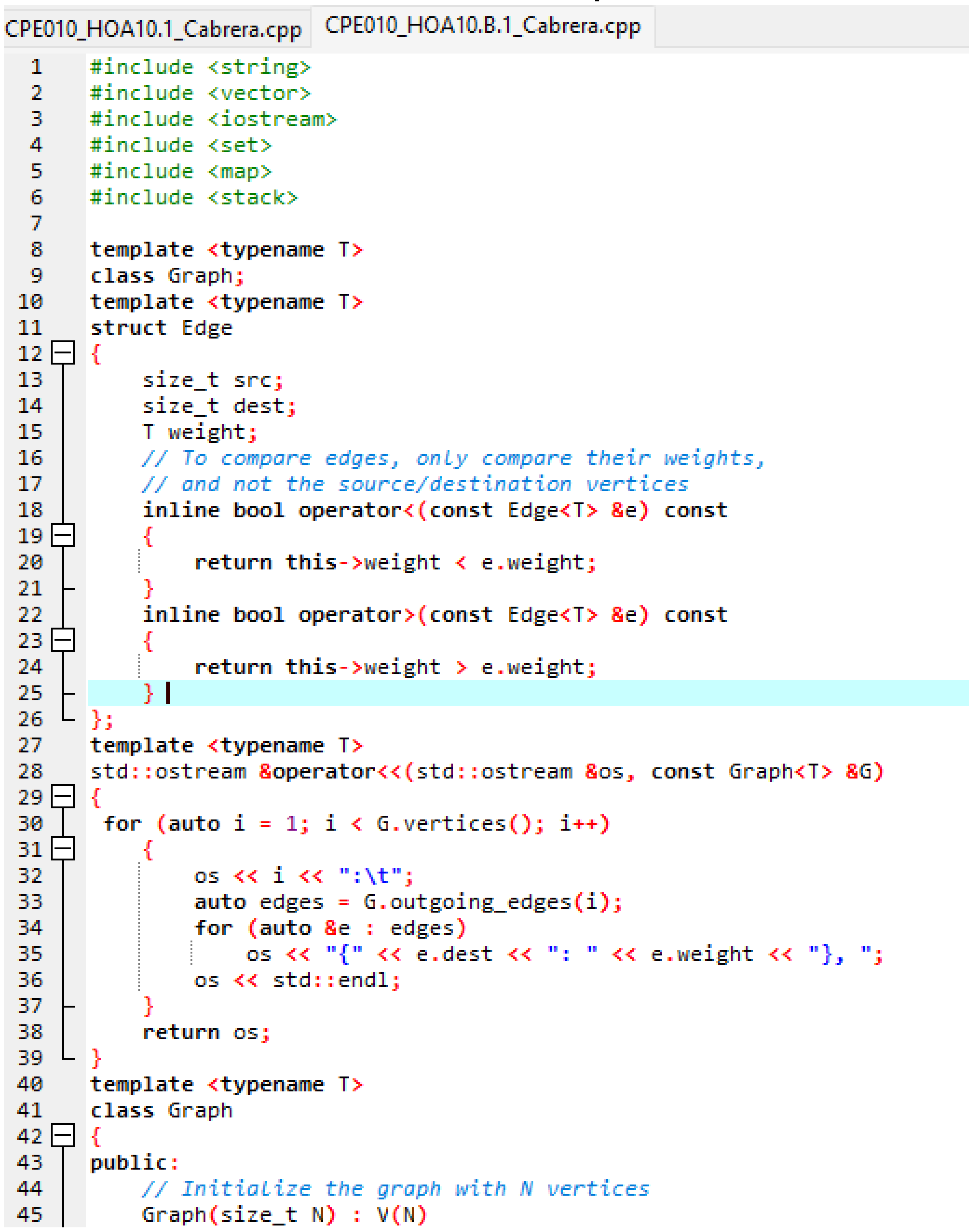
C:\Users\TIPQC\Desktop\CPE(

```
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 0.01193 seconds with return value 0
Press any key to continue . . .
```

ILO A - Graph Implementation Output

The output of the main function where it shows {0,2,4} first then {0,1,2} before displaying the rest of the list in the correct order.

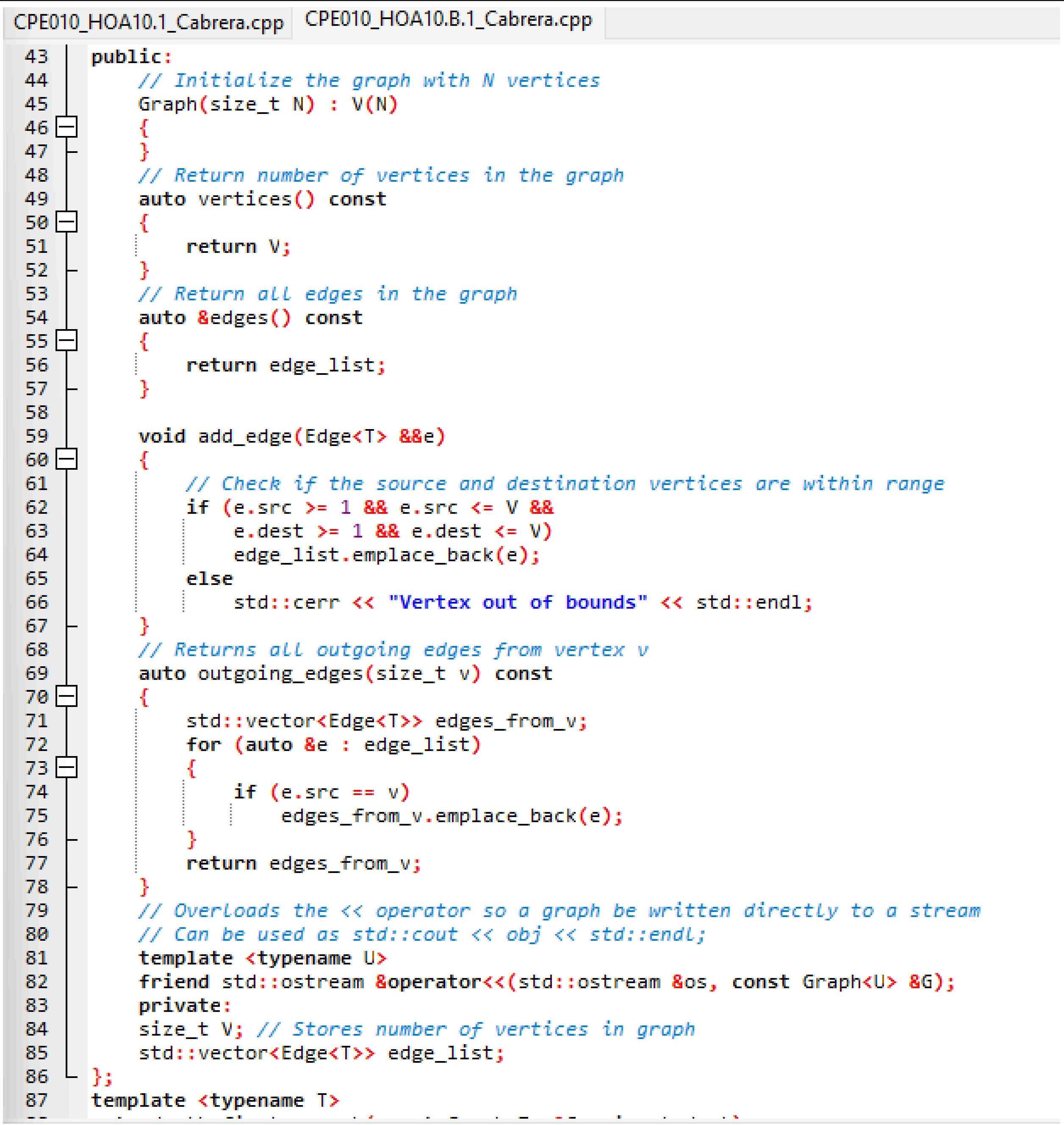
B.1. Depth-First Search



CPE010_HOA10.1_Cabrera.cpp CPE010_HOA10.B.1_Cabrera.cpp

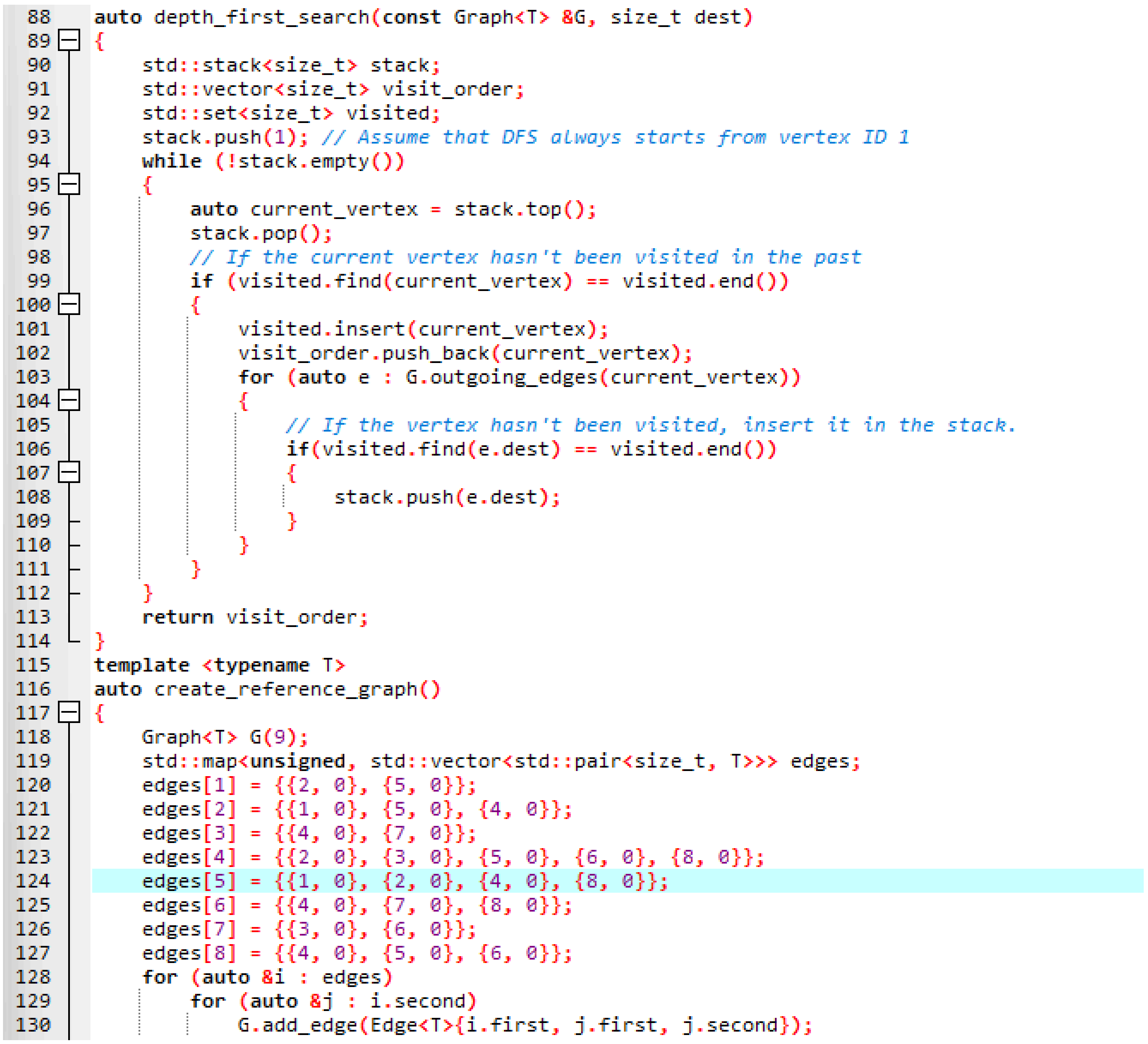
```cpp
#include <string>
#include <vector>
#include <iostream>
#include <set>
#include <map>
#include <stack>

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)
```

The implementation of the graph in this code is similar to the previous one where the Edge structure holds the source, destination, and the weight to be used when showing how the vertices are connected. They all use template <typename T> so the whole program can be dynamic in handling various types of data.


CPE010_HOA10.1_Cabrera.cpp CPE010_HOA10.B.1_Cabrera.cpp

```cpp
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>
```

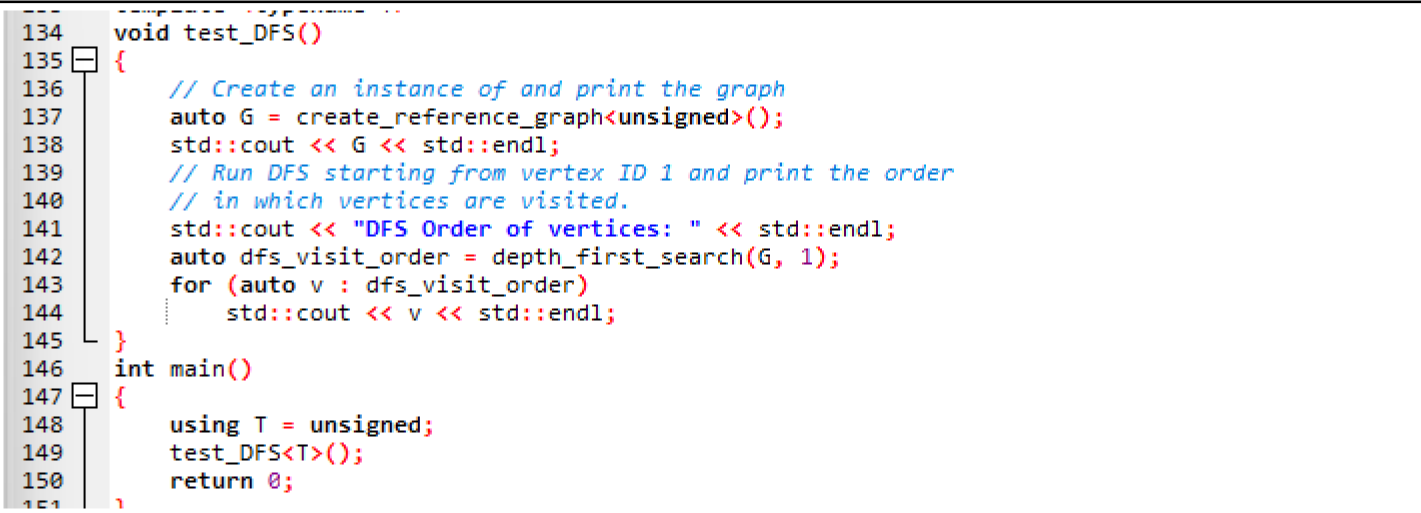
The Graph class holds the auto initialization of the vertex and edges of the graph, which also counts the number of vertices and edges of the graph. It also holds the add_edge and outgoing_edge functions to check if the source and destination vertices are in range and returns all the outgoing edges from the vertex called.



```
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});
```

This depth_first_search function is the core of this algorithm, where it uses the stack library to find the deepest parts of the graph. It first starts by checking all the connected vertices from the source vertex and adds them to the stack, it will then check if any vertex is connected to the first vertex searched, if there is any, it will add those vertices to the stack and will continue this until there are no connected vertex is seen anymore, the program will then start popping the stack and will continue checking the other vertices until it finishes searching the whole graph.

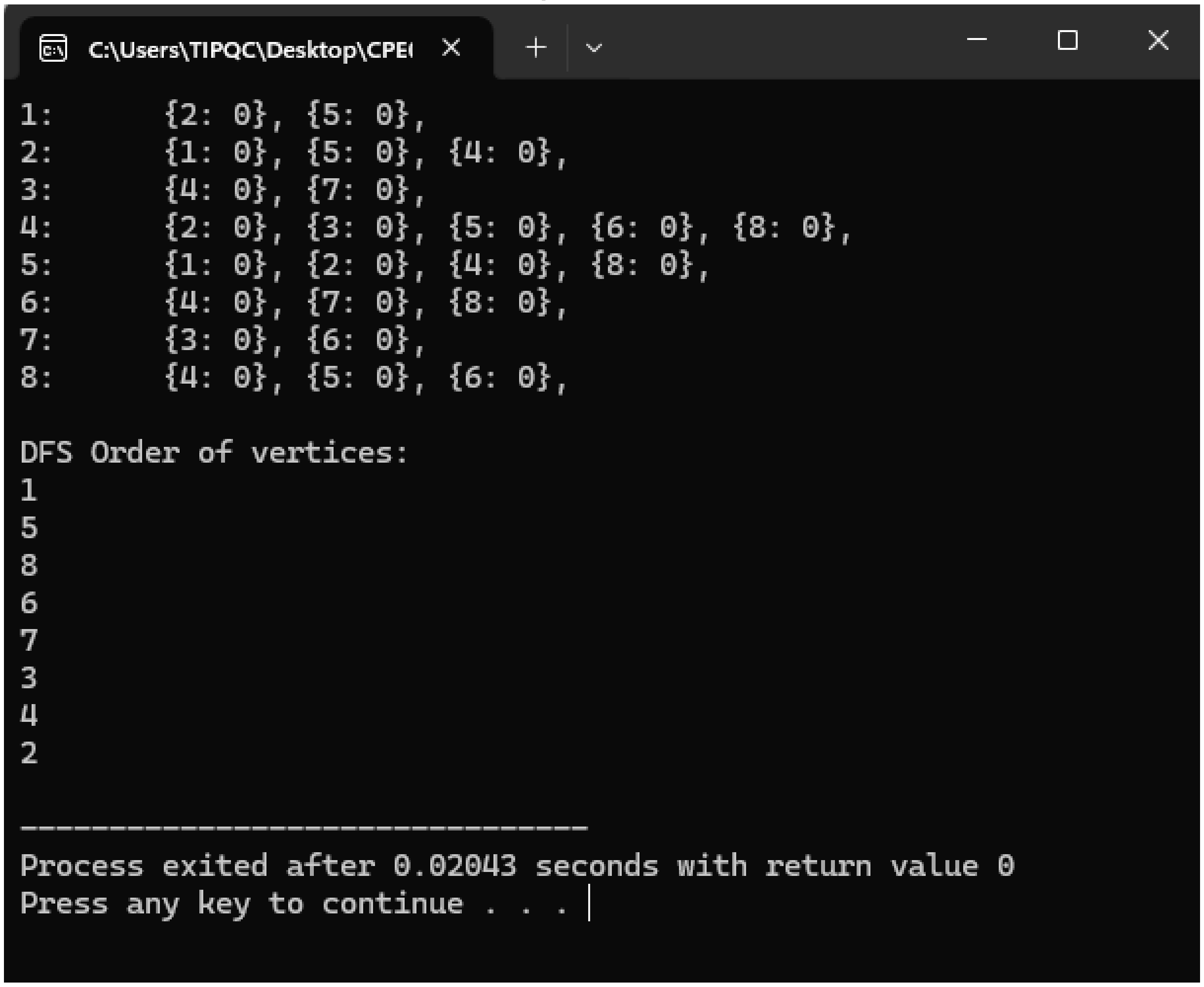
The create_reference_graph will be the one to handle creating the graph which the depth_first_search function will be operating on.



```
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;
```

The rest of the code will just combine all the previously made ones to show how the depth-first search algorithm is implemented.

ILO B.1 - Depth-First Search Code

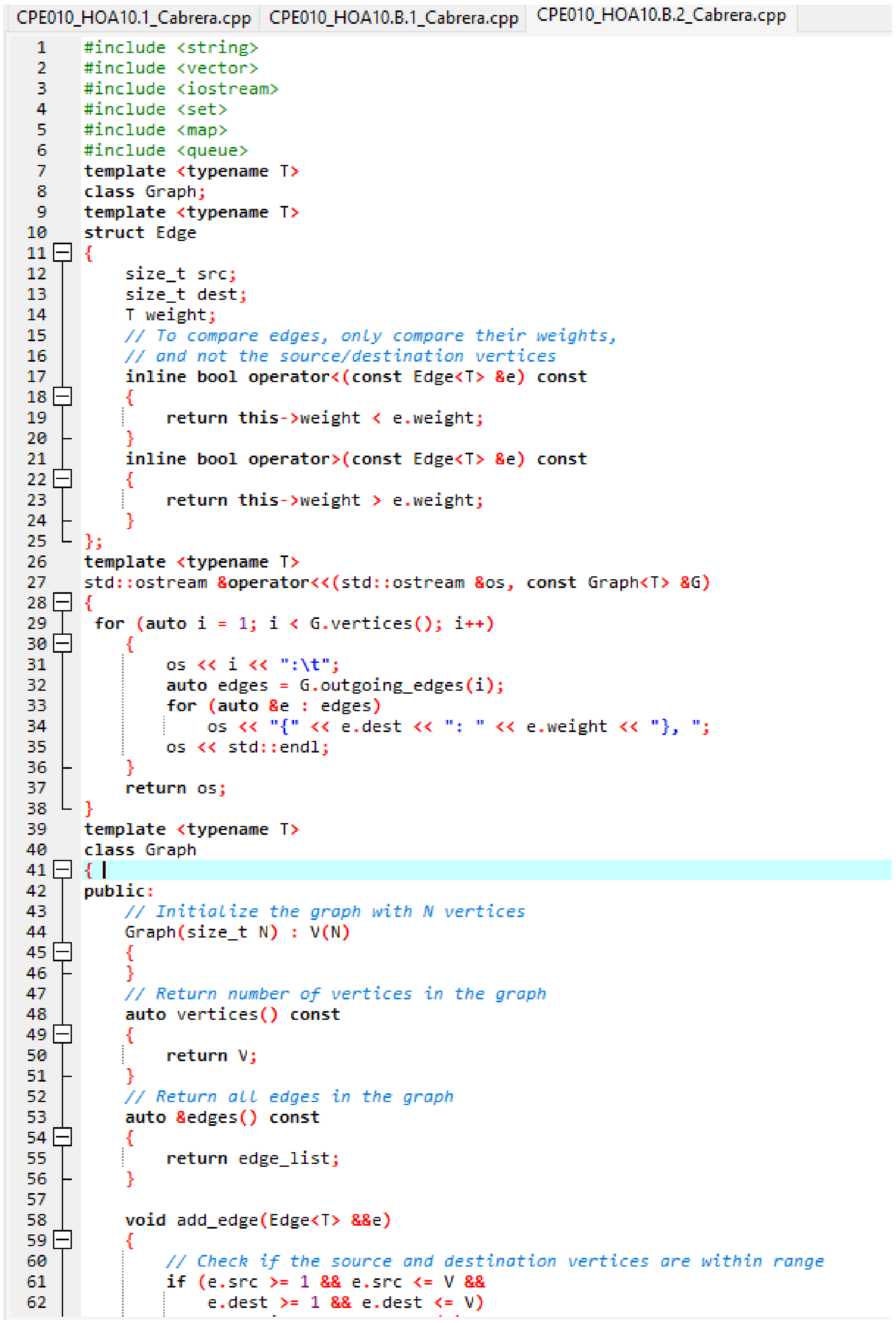


```
1:      {2: 0}, {5: 0},
2:      {1: 0}, {5: 0}, {4: 0},
3:      {4: 0}, {7: 0},
4:      {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0},
5:      {1: 0}, {2: 0}, {4: 0}, {8: 0},
6:      {4: 0}, {7: 0}, {8: 0},
7:      {3: 0}, {6: 0},
8:      {4: 0}, {5: 0}, {6: 0},

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

--------------------------------
Process exited after 0.02043 seconds with return value 0
Press any key to continue . . .
```

ILO B.1 - Depth-First Search Output

The top half shows the initialization of the graph, the first number representing the number in the node while the second one is the weight of the node, which is 0 in this case since depth-first doesn't require weights in searching. The bottom half shows how the algorithm traversed through the given graph.

B.2. Breadth-First Search

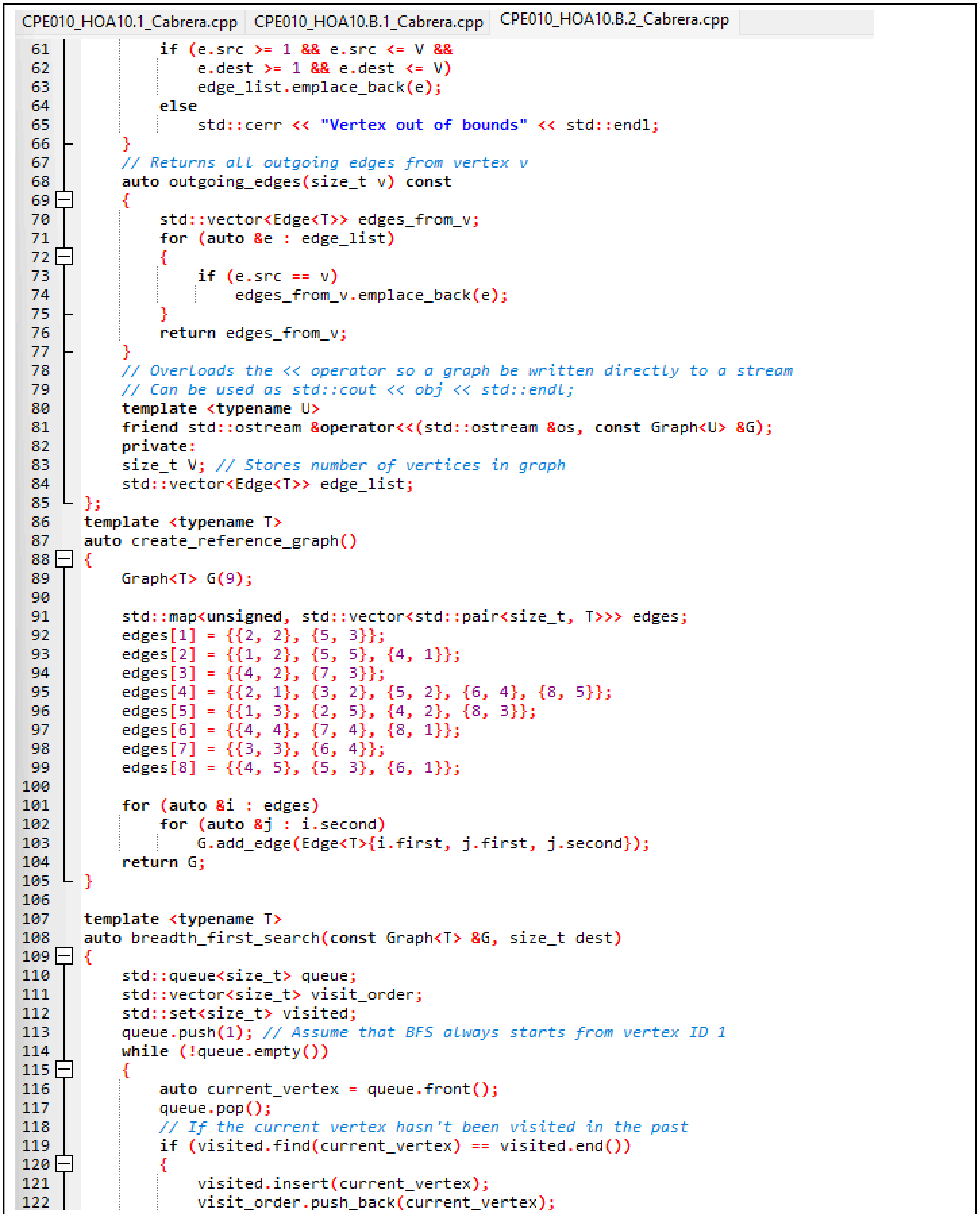


CPE010_HOA10.1_Cabrera.cpp CPE010_HOA10.B.1_Cabrera.cpp CPE010_HOA10.B.2_Cabrera.cpp

```
#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;
    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)
```

The implementation of the graph in this code is exactly like the previous one where the Edge structure holds the source, destination, and the weight to be used when showing how the vertices are connected. They all use template <typename T> so the whole program can be dynamic in handling various types of data.

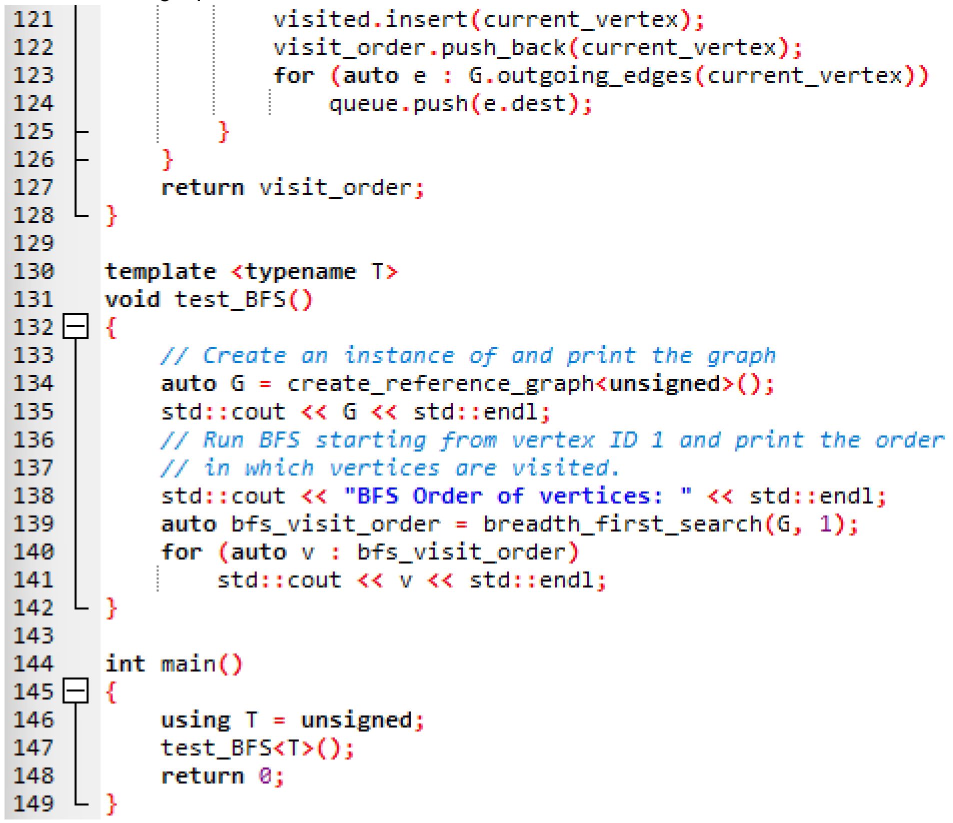
The Graph class holds the auto initialization of the vertex and edges of the graph, which also counts the number of vertices and edges of the graph. It also holds the add_edge and outgoing_edge functions to check if the source and destination vertices are in range and returns all the outgoing edges from the vertex called.


CPE010_HOA10.1_Cabrera.cpp CPE010_HOA10.B.1_Cabrera.cpp CPE010_HOA10.B.2_Cabrera.cpp

```cpp
        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 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);
```

The create_reference_graph will be the one to handle creating the graph which the breadth_first_search function will be operating on.

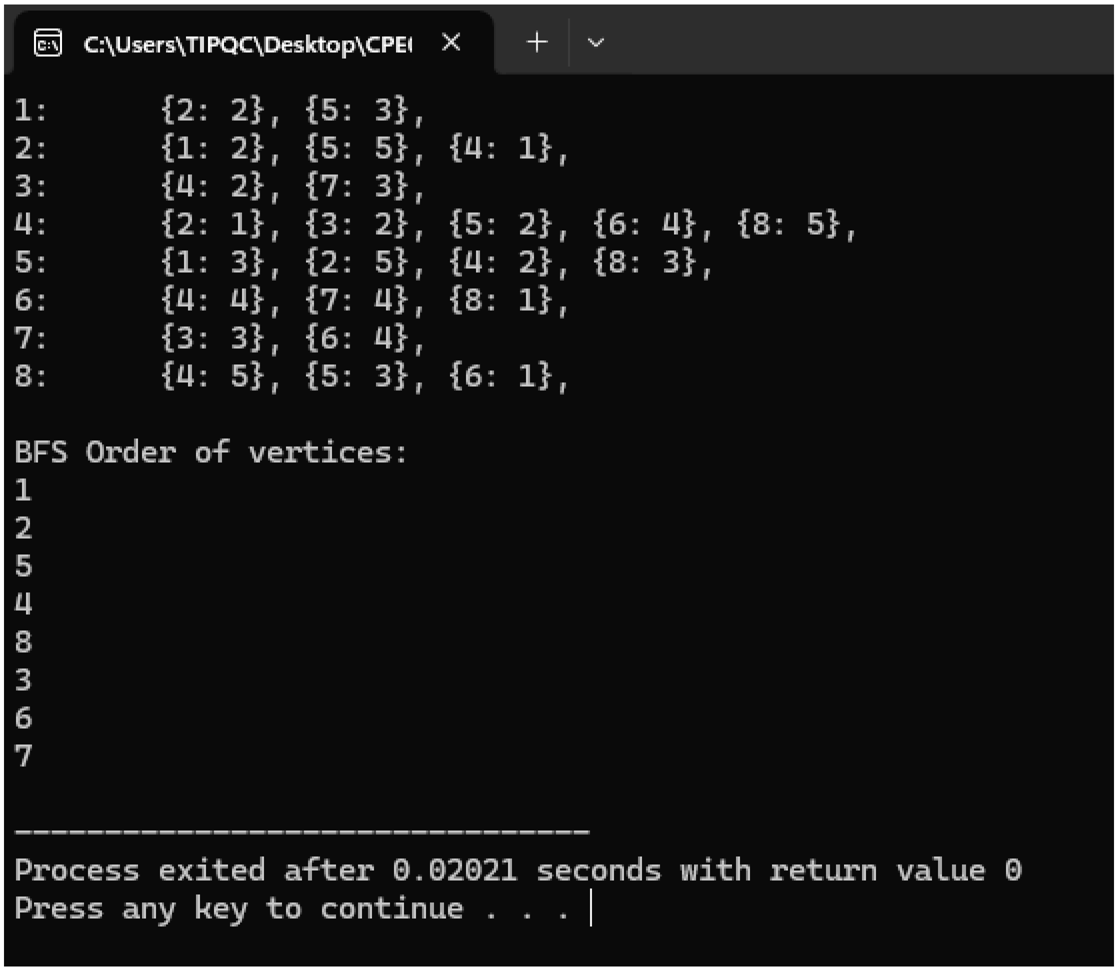
Opposite to the depth-first search, the breadth-first search first searches the upper layers of the graph using queue implementation, where the algorithm adds the connected vertices of the vertex to a queue instead of a stack, this way, whichever node the program sees first will be the first one to be searched, doing a left to right then top to bottom search for the whole graph.



```cpp
            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;
}
```

The rest of the code will just combine all the previously made ones to show how the breadth-first search algorithm is implemented.

ILO B.2 - Breadth-First Search Code



```
C:\Users\TIPQC\Desktop\CPE(

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 0.02021 seconds with return value 0
Press any key to continue . . .
```

ILO B.2 - Breadth-First Search Output

The top half shows the initialization of the graph, the first number representing the number in the node while the second one is the weight of the node, while there are assigned weights in the graph,breadth-first search doesn't require weights in searching since it just executes a top to bottom and left to right search approach. The bottom half shows how the algorithm traversed through the given graph.

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

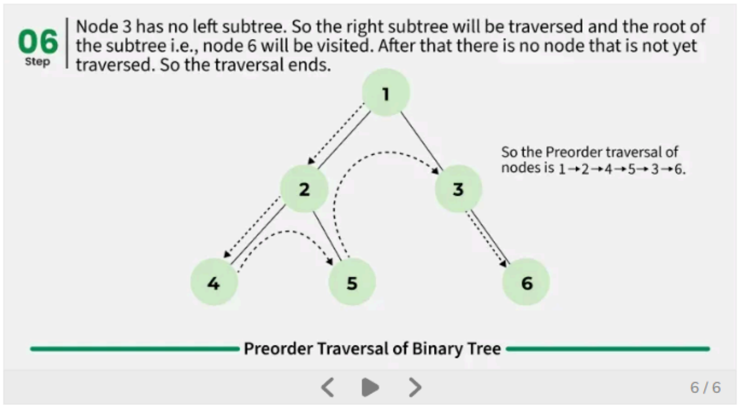
<u>This is a depth-first search, where the algorithm goes to the deepest part of the map before going back and doing the same to the rest of the nodes.</u>

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.

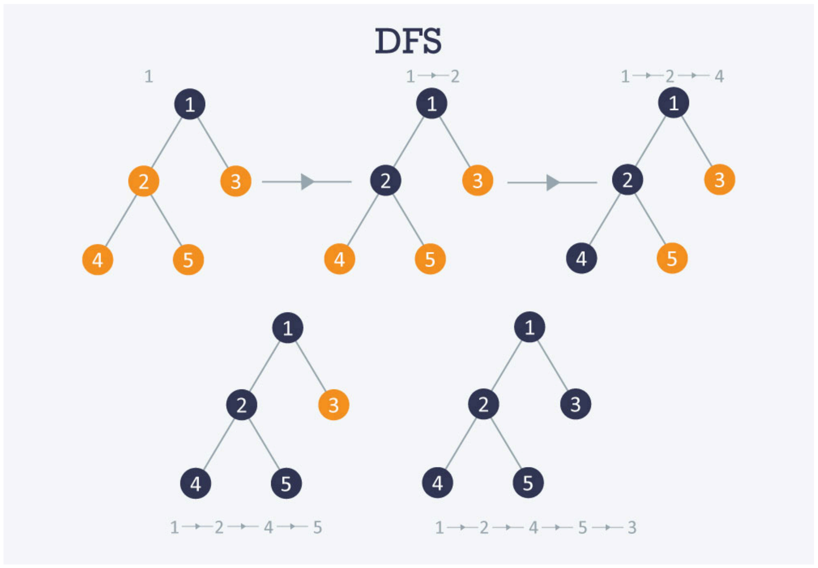
<u>The depth-first search is similar to preorder traversal where the algorithm visits every left subtree from the root node then goes right from there until it visits all the nodes.</u>

2. Preorder Traversal

- *Visit the root*
- *Traverse the left subtree, i.e., call Preorder(left-subtree)*
- *Traverse the right subtree, i.e., call Preorder(right-subtree)*


06 Step | Node 3 has no left subtree. So the right subtree will be traversed and the root of the subtree i.e., node 6 will be visited. After that there is no node that is not yet traversed. So the traversal ends.

So the Preorder traversal of nodes is 1→2→4→5→3→6.



3. In the performed code, what data structure is used to implement the Breadth First Search?
Queue is used for the Breadth-First search in the performed code, this is perfect because the queue visits the first ones in queue first, then adds more to the back whenever the algorithm finds more nodes, then goes one by one to traverse all the discovered nodes.
4. How many times can a node be visited in the BFS?
Only once, regardless of how many edges are connected to it.

8. Conclusion

In this activity, I have learned how to create a graph and implement it to code. I have also learned how to traverse these graphs using breadth-first and depth-first algorithms, discovering their difference and which data structure to use in implementing a search. I have also learned which one to use depending on the priority of the data I am searching. Overall, this has been useful knowledge to implement in future projects and activities.

9. Assessment Rubric