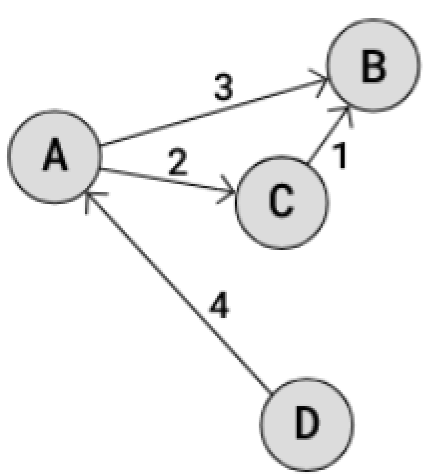
Quiz 10.1	
Graph	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/02/2025
Section: CPE21S4	**Date Submitted:** 10/02/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

6. Output

1. Define the following: (10)

A. **Graph** - A graph is a data structure similar to trees, where it consists of nodes connected to each other, the difference is this one is non-hierarchical, which means there are no root nodes, parent nodes, or children nodes. Nodes in a graph can be interconnected with one another and can be connected with more than one node.
B. **Vertex** - Basically the points or nodes in the graph, these are connected with lines called edges.
C. **Nodes** - Nodes are the other term for vertices, these are also connected by edges and can be interconnected with more than one other node.
D. **Weight** - The cost of data passing through a path, these can be in bandwidth, memory, or other types of cost.
E. **Path** - A path shows the flow of the interconnectivity of a graph, it starts at an origin graph and ends if there is no more vertex connected to the last node.
F. **Directed graph** - A graph where the edges have arrows that symbolize how the data passes through each node, for example, A graph having A->B nodes can have data passing from A to B but not B to A.
G. **Connected graph** - Connected graphs are two graphs that are connected with one another through one of their nodes.
H. **Directed cyclic** - A directed cyclic graph is a graph with arrows and interconnected nodes, showing how data can pass through them, eventually leading or circling back to an origin or reference node.
I. **Adjacency matrix** - An adjacency matrix shows the connectivity of the matrix in a tabled manner, where the intersections of the nodes are where the edges are represented, together with their weight or cost if there are any.
J. **Adjacency List** - This is the list version of the paths in a graph where every vertex is shown along with all the vertices that they are adjacent or connected with.

2. Identify the parts of the following graph. (10)



A ,B, C, D -Vertices
1,2,3,4-Edges
E1 = {C,B,1}

E2 = {A,C,2}
E3 = {A,B,3}
E4 = {D,A,4}

3. Plot the adjacency matrix of the graph above. (10 pts)

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

4. Create a program in C++ program to create a graph that looks like in the above figure.

main.cpp

Share

Run

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

main.cpp

Run

```cpp
    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
```

main.cpp

```cpp
        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;
    }

    auto outgoing_edges(size_t v) const
    {
        std::vector<Edge<T>> edges_from_v;
```

main.cpp

Run

```cpp
    {
        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;
```

```
1:
2:
3:
4:

DFS
1
2
3
BFS
1
2
3

===
```

main.cpp

Run

```cpp
<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
```

main.cpp

Run

```
(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 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);
    while (!queue.empty())
    {
        auto current_vertex = queue
```

main.cpp

Run

```cpp
    while (!queue.empty())
    {
        auto current_vertex = queue.front();
        queue.pop();
        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>
auto create_reference_graph()
{
    Graph<T> G(5);
    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
    edges[1] = {{1,1}, {2, 1}};
```

main.cpp

Run

```
        <std::pair<size_t, T>>>
        edges;
    edges[1] = {{1,1}, {2, 1}};
    edges[2] = {{2, 2}, {3, 3}};
    edges[3] = {{1, 3}, {2, 3}};
    edges[4] = {{4, 4}, {1, 4}};
    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()
{
    auto G = create_reference_graph
        <unsigned>();
    std::cout << G << std::endl;
    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;

}

template <typename T>
void test_BFS()
```

main.cpp

Run

```cpp
<unsigned>();
    std::cout << G << std::endl;
    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;

}

template <typename T>
void test_BFS()
{
    auto G = create_reference_graph<unsigned>();
    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_DFS<T>();
    test_BFS<T>();
    return 0;
}
```

```
Output
1:  {1: 1}, {2: 1},
2:  {2: 2}, {3: 3},
3:  {1: 3}, {2: 3},
4:  {4: 4}, {1: 4},

DFS Order of vertices:
1
2
3
BFS Order of vertices:
1
2
3
```

5. Differentiate a BFS vs DFS types of search.
The breadth-first search first searches the upper layers of a 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.

On the other hand, depth first search uses a stack 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.

7. Assessment Rubric