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): Francis Nikko I. Andoy	**Instructor:** Jimlord Quejado

6. Output

ILO A:

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
```

ANALYSIS:

In this code the first thing is that we created a node that is pointing to the next, we used struct so the data here is public by default. Since we are creating a graph we need an edge and we used a struct again to store all that. As for the class in private modifier this includes all the logic in the programming, like the dynamic memory allocation. Then the public head will have 2 pointers, as for the constructor this will iterate the graph since it has a 2 for loop inside of the constructor. Also the allocating will also happen inside of the constructor to initialize their value. After this we will use the destructor to delete all of the data to clean the memory and avoid memory leak. Next is just the display function and the main which is where all the value will come from.

In this given value
(0, 2, 4)
since we have a tree value, this actually indicate the: start of the vertex which is 0, the end of the vertex which is the 2, and the weight which is the 4.

```
        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;
}
```

OUTPUT:

```
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)
```

ILO B:

CODE:

```
#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;
```

ANALYSIS:

This code is focused on the DFS algorithm where stack is the most efficient way to store the data. The program begins by defining an edge struct that represents a connection between two vertices with a weight, and includes comparison operators based solely on the weight. The Graph class stores a list of edges and the total number of vertices, and provides methods to add edges, retrieve outgoing edges from a vertex, and print the graph.

```
    }
    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)
            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 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});
   return G;
}

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;
```

}

OUTPUT:

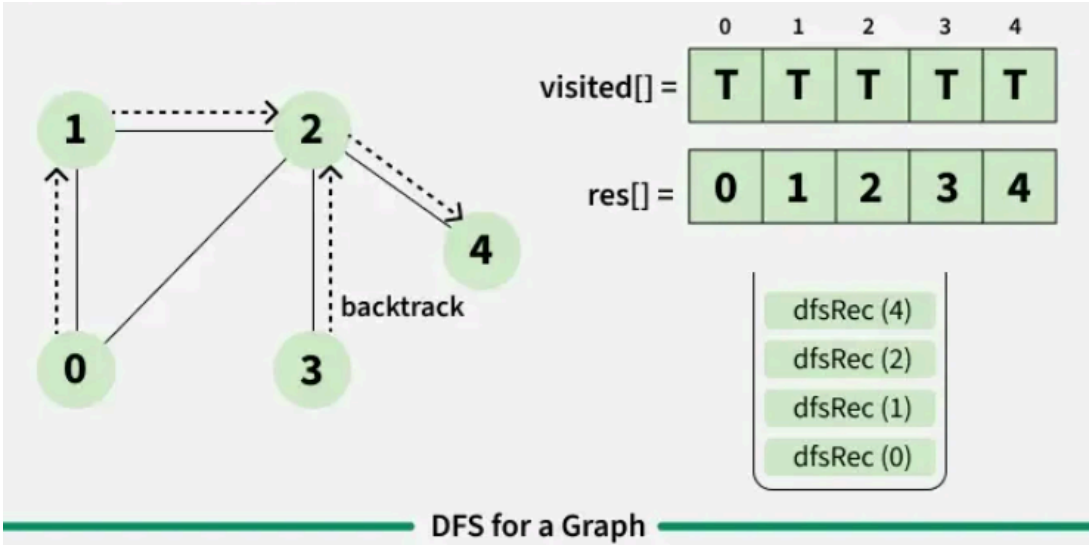
```
1:  {2: 0}, {5: 0},
2:  {1: 0}, {5: 0}, {4: 0},
3:  {4: 0}, {7: 0},
4:  {2: 0}, {3: 0}, {5: 0}, {6: 0}, {8: 0},
5:  {1: 0}, {2: 0}, {4: 0}, {8: 0},
6:  {4: 0}, {7: 0}, {8: 0},
7:  {3: 0}, {6: 0},
8:  {4: 0}, {5: 0}, {6: 0},

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

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 most suitable algorithm is the Depth-First Search or DFS because of the back tracking this algorithm explores the vertices 1 by 1 starting from the starting vertex, it moves in the most left side of the graph. Just like the name it will not stop until the last vertex is not reached on the left side. Backtracking happens when the last vertex in the left side is reached, it will backtrack to the parent node if there is a parent node then check if it has another child aside from the left side child. This will keep on happening until all the vertices are visited. This is also a stack data structure wherein the rule is LIFO, last in first out this is the basic rule when using stack, in DFSwe also add this kind of data structure. According to (GeeksforGeeks, 2025c), this is the visual presentation on how the DFS really works.



This is how backtracking works, it will go back to the parent node then check if there is still a child left if there's none left then backtrack again to the starting vertex.

GeeksforGeeks. (2025c, September 2). Depth first search or DFS for a graph. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/

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 corresponds to the 2 main strategies of the traversal strategies for trees, these are the preorder and post order. Preorder visits the root first, then the left and right subtrees. Postorder visits the left and right subtrees before the root. In nature these 2 is just a DFS since they all start from the starting node then moves to the left side, again and again til it reaches the last left node. The traversal order depends on the application, such as expression evaluation or tree reconstruction. Preorder is useful for copying trees, while postorder is used in deleting trees.

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

- queue is the data structure used for the BFS, this will ensure that all nodes will be traversed in an exact way. Like layer by layer traverse and the current layer will be the first visited, this kind of data structure the queue is the right choice. The queue is initialized with the starting vertex, and as each vertex is processed, its unvisited neighbors are enqueued. This guarantees that the shortest path to any reachable vertex is found. Additionally by using the queue data structure we can avoid revisiting the nodes that is why we can only visit them once then it is done. The traversal is done when it reaches the last layer and the last node of that layer.

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

- Just once since the BFS will move layer by layer then at the first layer it will traverse all the nodes in that same current layer. After that if there is still a layer it will again move to the left side of that layer then traverse all of the nodes of that current layer. So basically as long as there is still a layer the it will just keep on moving horizontally. This also includes the queue data structure which follows the rule FIFO, first in first out

8. Conclusion

- In conclusion, this graph traversal is an essential in data structure because of how efficient this really is. Just like these 2 Depth First Search (DFS) and Breadth First Search (BFS) these 2 serve as foundational techniques for navigating graphs, each with distinct strategies and use cases. For the DFS this backtracks the nodes. Backtracking happens when the last vertex in the left side is reached, it will backtrack to the parent node if there is a parent node then check if it has another child aside from the left side child. This will keep on happening until all the vertices are visited. Wherein the BFS will just visit all the nodes once this will move to the left side then traverse all the nodes that are inside of that layer. This will keep on happening until the last layer and the last node is reached. The adjacency list representation, as demonstrated in both code examples, offers an efficient way to store sparse graphs and supports fast edge insertion and traversal. Moreover, the analysis of these codes emphasizes the practical application of graph theory in software development. From printing adjacency lists to performing DFS, each function showcases how abstract concepts translate into executable logic.

9. Assessment Rubric