Activity No. 10.2	
Hands-on Activity 10.2 Implementing DFS and Graph Applications	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** November 27, 2024
Section: CPE21S4	**Date Submitted:** November 27, 2024
Name(s): - ESTEBAN, Prince Wally G. - FERNANDEZ, Don Eleazar T. - SANCHEZ, Christan Ray R. - TANDAYU, Leoj Jeam B. - VALLESER, Kenn L.	**Instructor:** Engr. Maria Rizette Sayo
A. Output(s) and Observation(s)	

SOURCE CODE:

```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)
```

```cpp
            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 T>    // removed this since it was already declared at the start of the this entire functon...
    friend std::ostream &operator<<<>(std::ostream &os, const Graph<T> &G);

private:
```

```cpp
    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 (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 (visited.find(e.dest) == visited.end())   // changed stack.if to if since there is not function that exists like that
under stack...
                {
                    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;
}
```
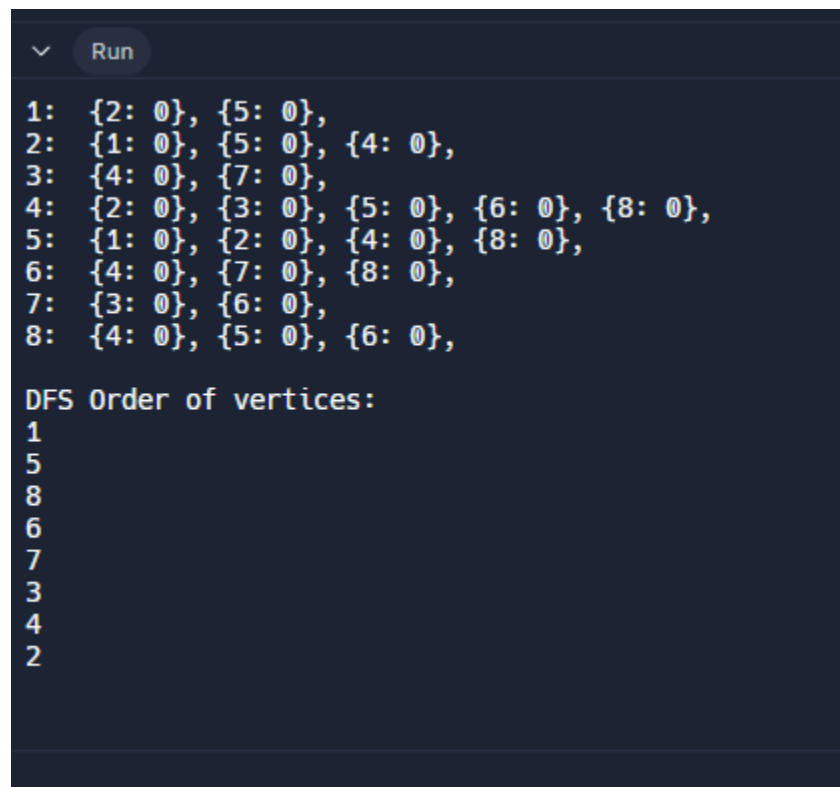
```cpp
template <typename T>
void test_DFS()
{
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;
    std::cout << "DFS Order of vertices: " << std::endl;
    auto dfs_visit_order = depth_first_search(G, 1);
    for (auto v : dfs_visit_order) {
        std::cout << v << std::endl;
    }
}

int main()
{
    using T = unsigned;
    test_DFS<T>();
    return 0;
}
```

OUTPUT:

```
 ∨   Run

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
```

OBSERVATION:

We observed that the code implements a depth-first search (DFS) on a graph using an adjacency list representation, with traversal managed by a stack. The DFS algorithm starts from vertex 1 and visits vertices in a specific order, keeping track of visited nodes with a set. The graph is undirected, and while edges have weights, these weights are not utilized in the DFS traversal. The DFS function only visits vertices reachable from vertex 1, limiting its use in disconnected graphs. A possible improvement could be to allow DFS to start from any vertex and incorporate edge weights for algorithms that require them.

B. Answers to Supplementary Activity

Answer the following questions:
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 visiting all vertices and backtracking is "Depth-First Search (DFS)". It starts at one vertex, explores as far as possible along each path, and backtracks when necessary to explore other paths from the same vertex.

2. Describe a situation where in the DFS of a graph would possibly be unique.

- DFS would be unique in cases where the graph is a tree, because there is only one way to traverse from the root to any leaf. The traversal order is determined by the starting point and the chosen direction (left or right).

3. Demonstrate the maximum number of times a vertex can be visited in the DFS. Prove your claim through code and demonstrated output.

- In DFS, a vertex can be visited multiple times if there are cycles in the graph or if revisiting vertices is allowed. For example, in a graph with a cycle, a vertex can be visited more than once.

4. What are the possible applications of the DFS?

- DFS is used in many areas, including solving mazes, finding connected components in a graph, detecting cycles, topological sorting in scheduling, and solving puzzles like Sudoku.

5. Identify the equivalent of DFS in traversal strategies for trees. In order to efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.

- The equivalent of DFS in tree traversal is "preorder, inorder, and postorder" traversal. All these methods use DFS to visit nodes in different orders (before, between, or after child nodes).

C. Conclusion & Lessons Learned

In conclusion, **Depth-First Search (DFS)** is an effective algorithm for exploring all vertices of a graph, especially when backtracking is required. It is particularly useful in situations like tree traversal, where the order of visiting nodes is essential. DFS can be unique when the graph has a simple structure, such as a tree, where each path is distinct. Its ability to revisit nodes in graphs with cycles makes it versatile for various applications like cycle detection and maze solving. Overall, DFS plays a crucial role in both graph and tree traversal, with different strategies like preorder, inorder, and postorder being its tree traversal equivalents.

D. Assessment Rubric

E. External References