

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: 11/27/2024
Section: CpE21S4	Date Submitted: 11/27/2024
Name(s): Virtucio, Dominic Joseph Bonifacio, Nyko Adrein Magistrado, Aira Pauleen Planta, Calvin Earl Solis, Paul Vincent	Instructor: Prof. Sayo

#### A. Output(s) and Observation(s)

##### Step 1:

```
10.2.cpp ×
1  #include <string>
2  #include <vector>
3  #include <iostream>
4  #include <set>
5  #include <map>
6  #include <stack>
7
8  template <typename T>
9  class Graph;
```

##### Step 2:

```
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:

```
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:

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

private:
    size_t V; // Stores number of vertices in graph
    std::vector<Edge<T>> edge_list;
};
```

### Step 5:

```
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))
            {
                stack.if (visited.find(e.dest) == visited.end())
                {
                    stack.push(e.dest);
                }
            }
        }
    }
    return visit_order;
}
```

### Step 6:

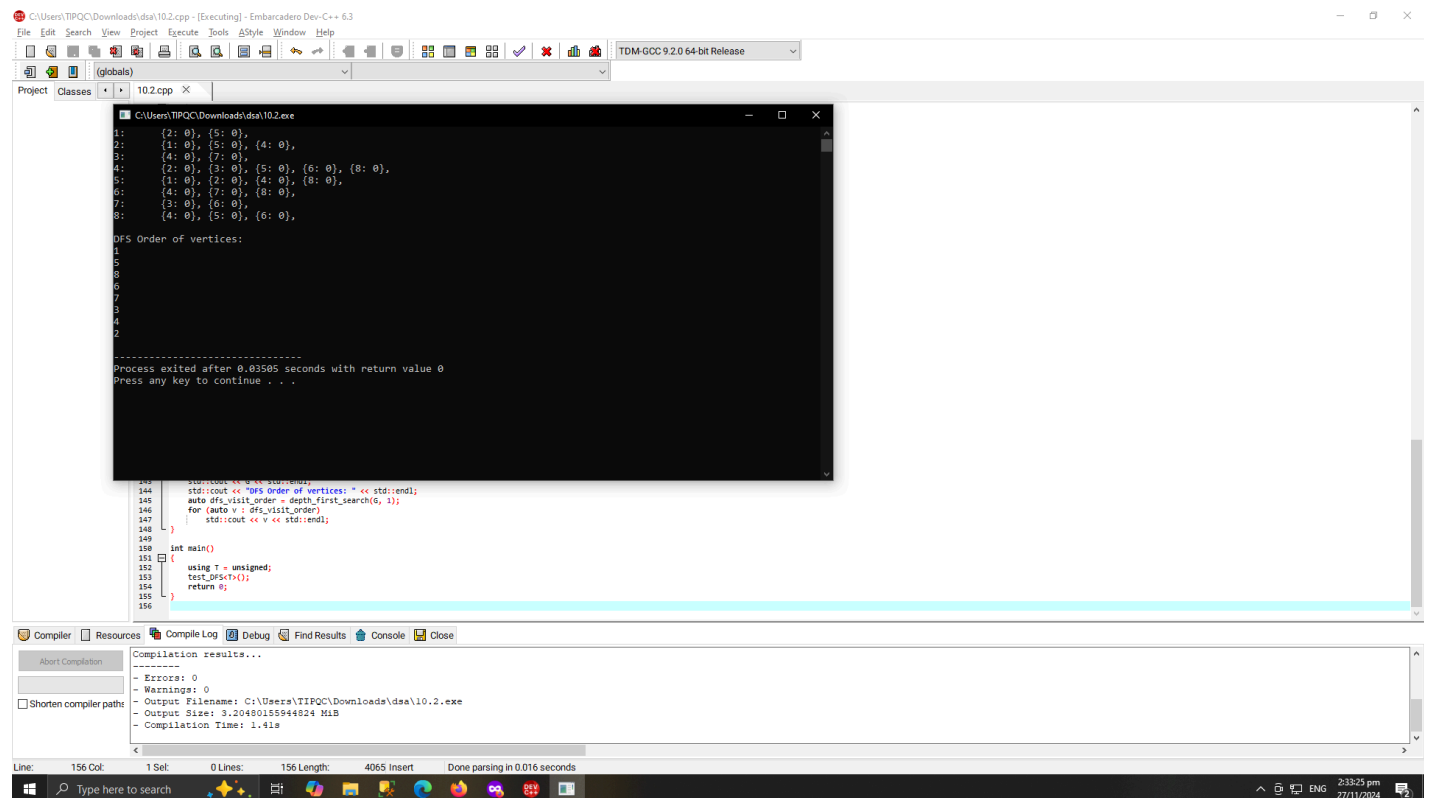
```
template <typename T>
auto create_reference_graph()
{
    Graph<T> G(9);
    std::map<unsigned, std::vector<std::pair<size_t, T>>> edges;
    edges[1] = {{2, 0}, {5, 0}};
    edges[2] = {{1, 0}, {5, 0}, {4, 0}};
    edges[3] = {{4, 0}, {7, 0}};
    edges[4] = {{2, 0}, {3, 0}, {5, 0}, {6, 0}, {8, 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;
}
```

### Step 7:

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

## OUTPUT:



```
C:\Users\TIPOC\Downloads\dsa\10.2.cpp - [Executing] - Embarcadero Dev-C++ 6.3
File Edit Search View Project Execute Tools Style Window Help
TDM-GCC 9.2.0 64-bit Release
Project Classes 10.2.cpp
C:\Users\TIPOC\Downloads\dsa\10.2.exe
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
0
7
3
4
2
-----
Process exited after 0.03505 seconds with return value 0
Press any key to continue . . .
137: std::cout << "DFS Order of vertices: " << std::endl;
144: auto dfs_visit_order = depth_first_search(1);
145: for (auto v : dfs_visit_order)
146:     std::cout << v << std::endl;
147: }
148:
149:
150: int main()
151: {
152:     using T = unsigned;
153:     test_DFS<T>();
154:     return 0;
155: }
156:
Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\TIPOC\Downloads\dsa\10.2.exe
- Output Size: 3,204,801,558+624 MB
- Compilation Time: 1.41s
Line: 156 Col: 1 Sel: 0 Lines: 156 Length: 4065 Insert Done parsing in 0.016 seconds
Type here to search 2:33:25 pm 2/11/2024
```

### Observation:

The corrected code represents a graph where each vertex is connected by weighted edges, and a Depth-First Search (DFS) is implemented to explore all vertices starting from vertex 1. The graph allows traversal by its edges, printing the order in which vertices are visited. The correction involved fixing a syntax error in the DFS function and ensuring the graph structure handles edges properly. This implementation serves as a good foundation for understanding graph traversal algorithms like DFS in C++, with a simple example graph.

## 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.
  - To accomplish the task of visiting every vertex on a map, a Depth-First Search (DFS) algorithm would be most helpful. The person starts at one vertex and explores as far as possible along each branch before backtracking. This ensures that all vertices are visited, and DFS will backtrack when it reaches a dead-end to explore other vertices, allowing for a complete traversal of the graph. DFS fits this scenario because it inherently handles backtracking and visiting all locations from the starting point.
2. Describe a situation where in the DFS of a graph would possibly be unique.
  - DFS might be unique when the graph is not fully connected. For example, in a directed graph with multiple disconnected subgraphs, the DFS will explore only one subgraph starting from the given vertex. The order in which vertices are visited can be unique depending on the starting vertex and the structure of the graph, particularly when there are cycles or paths that can be traversed in different ways. The DFS will also produce a unique visitation order based on the specific order of the edges being traversed.
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 is typically visited once when the search algorithm reaches it for the first time. However, if there are multiple calls or cycles in the graph, the vertex may be revisited. To demonstrate this with code, we

can simulate a graph where a vertex is part of a cycle or is revisited due to multiple recursive calls in the DFS. Here's an example:

```
C/C++
#include <iostream>
#include <vector>
#include <set>
#include <stack>

template <typename T>
class Graph {
public:
    Graph(size_t N) : V(N) {}

    void add_edge(size_t src, size_t dest) {
        adj_list[src].push_back(dest);
    }

    void depth_first_search(size_t start) {
        std::set<size_t> visited;
        dfs_recursive(start, visited);
    }

private:
    size_t V;
    std::vector<std::vector<size_t>> adj_list;

    void dfs_recursive(size_t vertex, std::set<size_t>& visited) {
        visited.insert(vertex);
        std::cout << "Visiting: " << vertex << std::endl;

        for (auto& neighbor : adj_list[vertex]) {
            if (visited.find(neighbor) == visited.end()) {
                dfs_recursive(neighbor, visited);
            }
        }
    }
};

int main() {
    Graph<int> G(5);
    G.add_edge(1, 2);
    G.add_edge(2, 3);
    G.add_edge(3, 4);
    G.add_edge(4, 1); // This creates a cycle

    std::cout << "Starting DFS from vertex 1:" << std::endl;
    G.depth_first_search(1); // Should demonstrate revisiting vertices
    return 0;
}
```

In this example, vertex 1 will be visited multiple times as part of the cycle. However, the standard DFS implementation ensures each vertex is only visited once in a typical traversal unless explicitly allowed for revisits or recursive exploration.

#### 4. What are the possible applications of the DFS?

- DFS has a wide range of applications. It can be used in solving problems that involve exploring all possibilities, such as in puzzle solving, pathfinding, and scheduling problems. It's useful for tasks like detecting cycles in graphs, checking graph connectivity, finding strongly connected components in directed graphs, and solving

mazes. Additionally, DFS is employed in topological sorting, and it can help in problems like finding all connected components in an undirected graph.

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 also Depth-First Search. Both DFS in graphs and trees explore as deeply as possible along each branch before backtracking. The main difference between DFS on a graph and a tree is that DFS in a tree doesn't need to worry about revisiting vertices, as trees don't have cycles. In terms of tree traversal, DFS can be implemented as pre-order, in-order, or post-order traversal, depending on the order in which the nodes are visited.

### **C. Conclusion & Lessons Learned**

In conclusion, Depth-First Search (DFS) is an effective method for traversing graphs and trees, allowing for thorough exploration of all vertices or nodes. It is especially useful in tasks such as maze solving, cycle detection, and finding connected components. While DFS is straightforward, its behavior can be unique depending on the structure of the graph, and it is versatile enough to be applied across various algorithms. Understanding DFS in both graphs and trees is essential for solving problems that require systematic exploration of all paths or possibilities.

The lesson learned from studying Depth-First Search (DFS) is that it is a powerful traversal technique for exploring all vertices or nodes in a graph or tree. By prioritizing deep exploration before backtracking, DFS ensures a complete search of paths. Its applications are vast, from solving puzzles to detecting cycles in graphs. Understanding how DFS operates helps in solving complex problems efficiently, especially when systematic exploration is needed.

### **D. Assessment Rubric**

### **E. External References**