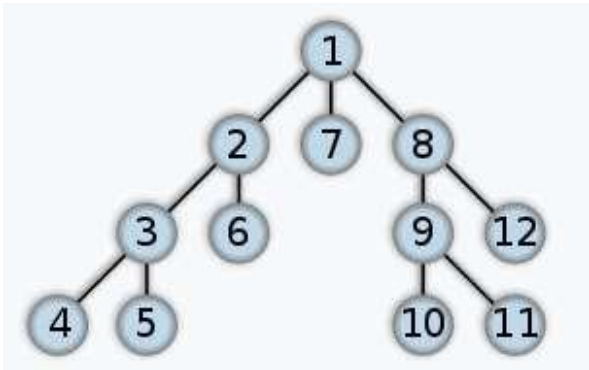


Activity No. 12	
Implementing Depth First Search	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed:
Section:	Date Submitted:
Name:	Instructor:
1. Objective(s)	
<ul style="list-style-type: none"> - Use the depth first search on graph data structures. - Solve given graph problems using depth first search. - Implement the different Depth First Search algorithms. 	
2. Intended Learning Outcomes (ILOs)	
After this activity, the student should be able to: <ul style="list-style-type: none"> - Implement the Depth First Search to solve a given problem. 	
3. Discussion	
<p>For a given vertex v, the depth-first search algorithm proceeds along a path from v as deeply into the graph as possible before backing up. That is, after visiting a vertex v, the depth-first search algorithm visits (if possible) an unvisited adjacent vertex to vertex v. The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v. We may visit the vertices adjacent to v in sorted order.</p>	
	
<p>The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.</p>	
<p>This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:</p> <ul style="list-style-type: none"> • Pick a starting node and push all its adjacent nodes into a stack. • Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. • Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop. 	
Recursive Depth-First Search Algorithm	
<pre>dfs(in v:Vertex) {</pre>	

```

// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
    Mark v as visited;
    for (each unvisited vertex u adjacent to v)
        dfs(u)
}

```

Iterative Depth-First Search Algorithm

```

dfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
    s.createStack();
    // push v into the stack and mark it
    s.push(v);
    Mark v as visited;
    while (!s.isEmpty()) {
        if (no unvisited vertices are adjacent to the vertex on
            the top of stack)
            s.pop(); // backtrack
        else {
            Select an unvisited vertex u adjacent to the vertex
                on the top of the stack;
            s.push(u);
            Mark u as visited;
        }
    }
}

```

4. Materials and Equipment

To properly perform this activity, the student must have:

- Personal Computer / Any device with C++ IDE
- C++ IDE must support at least C++11

5. Procedure

Step 1. Include the required header files, as follows:

```

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

template <typename T>
class Graph;

```

Step 2. Write the following struct in order to implement an edge in our 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;
    }
};

```

Step 3. Next, overload the << operator for the graph so that it can be printed out using the following function:

```

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. Implement the graph data structure that uses an edge list representation as follows:

```

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. Now, we need a function to perform DFS for our graph. Implement it as follows:

```

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();
    }
}

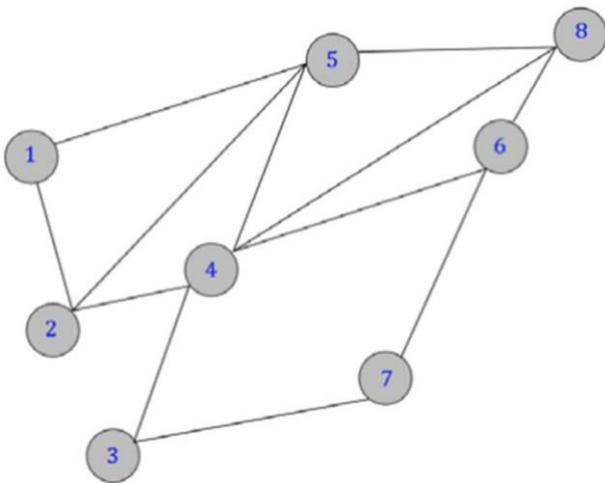
```

```

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

```

Step 6. We shall test our implementation of the DFS on the graph shown here:



Use the following function to create and return the graph:

```

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;
}

```

Note the use of null values for edge weights since DFS does not require edge weights. A simpler implementation of the graph could have omitted the edge weights entirely without affecting the behavior of our DFS algorithm.

Step 7. Finally, add the following test and driver code, which runs our DFS implementation and prints the output:

```

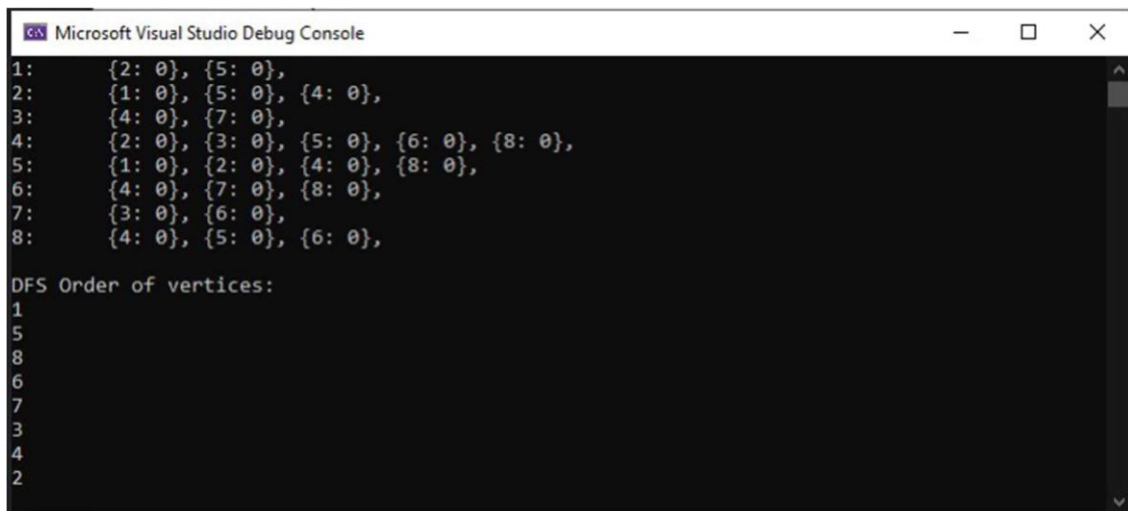
template <typename T>
void test_DFS()
{
    auto G = create_reference_graph<unsigned>();
    std::cout << G << std::endl;

    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;
}
```

Step 8. Compile and run the preceding code. Your output should look as follows:

A screenshot of the Microsoft Visual Studio Debug Console window. The window title is "Microsoft Visual Studio Debug Console". The output is as follows:

```
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
```

6. Output

7. 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.
2. Describe a situation where in the DFS of a graph would possibly be unique.
3. Demonstrate the maximum number of times a vertex can be visited in the DFS. Prove your claim through code and demonstrated output.
4. What are the possible applications of the DFS?
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.

8. Conclusion

9. Assessment Rubric

10. References

The following were used as resources for the development of this laboratory activity.

- <https://www.hackerearth.com/practice/algorithms/graphs/>
- <https://runestone.academy/runestone/books/published/cppds/>
- <https://user.ceng.metu.edu.tr/~ys/ceng707-dsa/>
- Anggoro, W. (2018). C++ Data Structures and Algorithms: Learn how to write efficient code to build scalable and robust applications in C++. Packt Publishing.