Lab 5: Implementation of graph traversal (BFS and DFS) using C++

Objective:

To implement and analyze two fundamental graph traversal algorithms: Breadth First Search (BFS) and Depth First Search (DFS). These algorithms are chosen to demonstrate different approaches to exploring graph structures and their applications in solving various computational problems.

Theory:

1. Breadth First Search(BFS): BFS explores a graph in a breadth wise motion, visiting all neighbors of a vertex before moving to the next level. It uses a queue data structure for finding shortest paths in unweighted graphs.

```
Algorithm:
Input: G (graph), start_vertex (starting point of traversal)
Output: Visited vertices in BFS order
a. Create a queue Q
b. Create a visited set
c. Add the start_vertex to visited
d. Enqueue start_vertex into Q
e. While Q is not emty
    i.   V = q.dequeue()
    ii.  Print v
    iii. For each neighbor w of v in G:
        a. If w is not in visited
        b. Add w to visited
        c. Enqueue w into Q
f. Return visited
```

2. Depth First Search (DFS): DFS explorers a graph by going as deep as possible along each branch before backtracking. It uses a stack for backtracking.

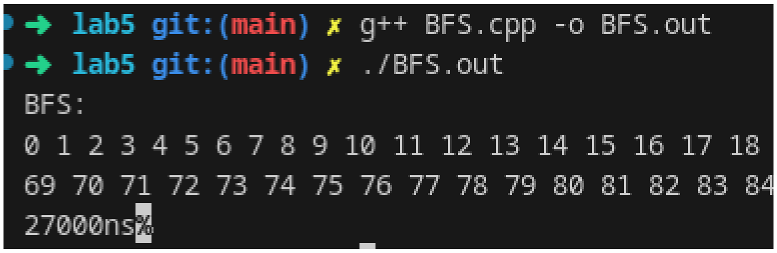
```
Algorithm:
Input : G (graph), start_vertex (starting point of traversal)
Output: Visited vertices in DFS order
a. Create a stack S
b. Create a visited set
c. Add start_vertex into S
d. While S is not empty:
    i.   V = S.pop()
    ii.  If V is not in visited
    iii. Add V to visited
    iv.  Print V
    v.   For each neighbor w of v in G:
        a. If w is not in visited:
        b. Push w into S
e. exit
```

Observation:

Breadth First Search

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>

using namespace std;

class Search {
    public:
    void bfs(vector<vector<int> >& adjList, int startNode,
            vector<bool>& visited)
    {
        queue<int> q;

        visited[startNode] = true;
        q.push(startNode);

        while (!q.empty()) {
            int currentNode = q.front();
            q.pop();
            cout << currentNode << " ";
            for (int neighbor : adjList[currentNode]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }

    void addEdge(vector<vector<int> >& adjList, int u, int v)
    {
        adjList[u].push_back(v);
    }
};

long long getTime(std::function<void()> f){
    auto start = clock();
    f();
    auto end = clock();
    long double duration = end - start;
    return (duration/CLOCKS_PER_SEC) * 1000000000;
}

int main()
{
    int vertices = 10;
    Search search;

    vector<vector<int> > adjList(vertices);

    for (int i = 0; i < vertices; i++) {
        search.addEdge(adjList, i, (i + 1) % vertices);
    }

    vector<bool> visited(vertices, false);

    auto bfs = [&](){
        search.bfs(adjList, 0, visited);
    };


    cout << "BFS: " << endl;
    cout << getTime(bfs);
    cout << "ns";
    return 0;
}
```

Output:

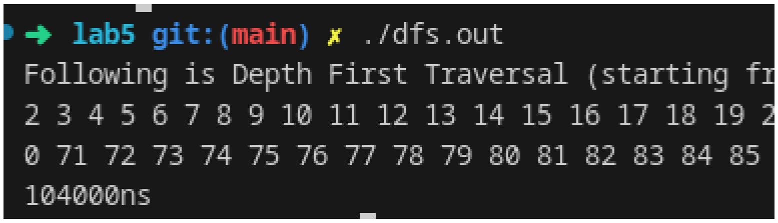


```
➜ lab5 git:(main) ✗ g++ BFS.cpp -o BFS.out
➜ lab5 git:(main) ✗ ./BFS.out
BFS:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
27000ns%
```

2. Depth First Search

```

#include <bits/stdc++.h>
using namespace std;

class Graph {
    public:
    map<int, bool> visited;
    map<int, list<int> > adj;
    void addEdge(int v, int w);
    void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::DFS(int v)
{
    visited[v] = true;
    cout << v << " ";

    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}

long long getTime(std::function<void()> f){
    auto start = clock();
    f();
    auto end = clock();
    long double duration = end - start;
    return (duration/CLOCKS_PER_SEC) * 1000000000;
}

int main()
{
    Graph g;
    for (int i = 0; i < 100; i++)
        g.addEdge(i, i + 1);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";

    auto dfs = [&](){
        g.DFS(2);
        cout << endl;
    };

    cout << getTime(dfs) << "ns Time taken" << endl;

    return 0;
}
```

Output:



```
➜ lab5 git:(main) ✗ ./dfs.out
Following is Depth First Traversal (starting fr
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 2
0 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
104000ns
```

Conclusion:

We implemented BFS and DFS using C++.