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:
         1. If w is not in visited
         2. Add w to visited
         3. Enqueuq 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:
    1. If w is not in visited:
    2. Push w into S

e. Exit

Observation

```
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
#include <string>
#include "gettime.h"
#define INFINITY -1

using namespace std;

template <class T>
bool vectorHas(vector<T> vec, T val)
{
    for (size_t i = 0; i < vec.size(); i++)
    {
        if (vec[i] == val)
        {
            return true;
        }
    }
    return false;
}

class Graph
{
public:
    vector<vector<int>> matrix;
    vector<string> names;
    int degree;
    Graph(int degree, vector<string> names)
    {
        this->degree = degree;
        this->names = names;
        matrix = vector<vector<int>>(degree, vector<int>(degree, 0));
    }

    vector<int> getAdjacent(int idx)
    {
        vector<int> arr;
        for (int i = 0; i < degree; i++)
        {
            if (matrix[idx][i] > 0)
            {
```

```
                arr.push_back(i);
            }
        }
        return arr;
    }

    void bfs()
    {
        queue<int> q;
        vector<int> visited;
        q.push(0);
        visited.push_back(0);
        do
        {
            int v = q.front();
            q.pop();
            // cout << names[v] << " " << endl;
            for (int i : getAdjacent(v))
            {
                if (!vectorHas(visited, i))
                {
                    q.push(i);
                    visited.push_back(i);
                }
            }
        } while (!q.empty());
    }

    void dfs()
    {
        stack<int> nodes;
        vector<int> visited;
        nodes.push(0);
        if (degree > 0)
        {
            while (visited.size() != degree)
            {
                int top = nodes.top();
                nodes.pop();
                if (!vectorHas(visited, top))
                {
                    // cout << names[top] << endl;
                    visited.push_back(top);
                }
```

```
                for (int j : getAdjacent(top))
                {
                    if (!vectorHas(visited, j))
                    {
                        nodes.push(j);
                    }
                }
            }
        }
    }
};

vector<string> names(int n)
{
    vector<string> res;
    for (int i = 0; i < n; i++)
    {
        res.push_back(to_string(i));
    }
    return res;
}

int main()
{
    vector<int> sizes;
    int start_size = 100;
    int increment = 100;
    sizes.push_back(start_size);
    srand(time(NULL));
    for (size_t i = 0; i < 4; i++)
    {
        sizes.push_back(sizes.back() + increment);
    }
    for (int size : sizes)
    {
        Graph g(size, names(size));
        vector<vector<int>> matrix;
        for (int i = 0; i < size; i++)
        {
            vector<int> row;
            for (int j = 0; j < size; j++)
            {
                if (rand() % 2 == 0)
                {
                    row.push_back(1);
```

```
                }
                else
                {
                    row.push_back(0);
                }
            }
            matrix.push_back(row);
        }
        g.matrix = matrix;

        cout << "size = " << size << endl;
        long long time = getTime([&]()
                                 { g.dfs(); });
        cout << "DFS time = " << time << endl;
        time = getTime([&]()
                       { g.bfs(); });
        cout << "BFS time = " << time << endl;
    }
}
```

Output:

```
size = 100
DFS time = 6083200
BFS time = 27629000
size = 200
DFS time = 23303700
BFS time = 334315500
size = 300
DFS time = 73125900
BFS time = 1657123700
size = 400
DFS time = 148268900
BFS time = 5651394000
size = 500
DFS time = 268807300
BFS time = 4536382200
```

Conclusion:

We implemented BFS and DFS using C++.