DFS-BFS

g++ -fopenmp -o output filename.cpp

./output

1

```
#include<iostream>
#include<omp.h>
#include<bits/stdc++.h>

using namespace std;

class Graph{
public:
    int vertices;
    vector<vector<int>> graph;
    vector<bool> visited;

    Graph(int v) : vertices(v), graph(v), visited(v, false) {}

    void addEdge(int a, int b){
        graph[a].push_back(b);
        graph[b].push_back(a);
    }

    void initialize_visited(){
        //visited.assign(vertices,false);

        for(int i=0;i<vertices;i++){
            visited[i]=false;
        }
    }

    void dfs(int i){
        stack<int> s;
        s.push(i);
        visited[i] = true;

        while(s.empty() != true){
            int current = s.top();
            cout << current << " ";
            s.pop();
            for(int j = 0; j < graph[current].size(); j++){
```

```
            int neighbor = graph[current][j];
            if(visited[neighbor] == false){
                s.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

void parallel_dfs(int i){
    stack<int> s;
    s.push(i);
    visited[i] = true;

    while(s.empty() != true){
        int current = s.top();
        cout << current << " ";
        #pragma omp critical
            s.pop();

        #pragma omp parallel for
        for(int j = 0; j < graph[current].size(); j++){
            int neighbor = graph[current][j];
            if(visited[neighbor] == false){
                #pragma omp critical
                {
                    s.push(neighbor);
                    visited[neighbor] = true;
                }
            }
        }
    }
}

void bfs(int i){
    queue<int> q;
    q.push(i);
    visited[i] = true;

    while(q.empty() != true){
        int current = q.front();
        q.pop();
        cout << current << " ";
        for(int j = 0; j < graph[current].size(); j++){
            int neighbor = graph[current][j];
            if(visited[neighbor] == false){
                q.push(neighbor);
                visited[neighbor] = true;
                }
            }
        }
    }

    void parallel_bfs(int i){
        queue<int> q;
        q.push(i);
        visited[i] = true;

        while(q.empty() != true){
            int current = q.front();
            cout << current << " ";
            #pragma omp critical
                q.pop();

            #pragma omp parallel for
            for(int j = 0; j < graph[current].size(); j++){
                int neighbor = graph[current][j];
                if(visited[neighbor] == false){
                    #pragma omp critical
                    {
                        q.push(neighbor);
                        visited[neighbor] = true;
                    }
                }
            }
        }
    }
};

int main()
{
    int vertices, edges;
    cout << "Enter the number of vertices: ";
    cin >> vertices;
    cout << "Enter the number of edges: ";
    cin >> edges;

    Graph g(vertices);

    for (int i = 0; i < edges; i++) {
        int a, b;
        cout << "Enter edge " << i + 1 << " (source destination): ";
        cin >> a >> b;
        g.addEdge(a, b);
    }
    int startNode;
    cout << "Enter the starting node for DFS and BFS traversals: ";
    cin >> startNode;

    cout << "Depth First Search: \n";
    auto start = chrono::high_resolution_clock::now();
    g.dfs(startNode);
    cout << endl;
    auto end = chrono::high_resolution_clock::now();
    cout << "Time taken: " << chrono::duration_cast<chrono::microseconds>(end
- start).count() << " microseconds" << endl;

    cout << "Parallel Depth First Search: \n";
    g.initialize_visited();
    start = chrono::high_resolution_clock::now();
    g.parallel_dfs(startNode);
    cout << endl;
    end = chrono::high_resolution_clock::now();
    cout << "Time taken: "<< chrono::duration_cast<chrono::microseconds>(end -
start).count() << " microseconds" << endl;

    cout << "Breadth First Search: \n";
    g.initialize_visited();
    start = chrono::high_resolution_clock::now();
    g.bfs(startNode);
    cout << endl;
    end = chrono::high_resolution_clock::now();
    cout << "Time taken: "<< chrono::duration_cast<chrono::microseconds>(end -
start).count() << " microseconds" << endl;

    cout << "Parallel Breadth First Search: \n";
    g.initialize_visited();
    start = chrono::high_resolution_clock::now();
    g.parallel_bfs(startNode);
    cout << endl;
    end = chrono::high_resolution_clock::now();
    cout << "Time taken: " << chrono::duration_cast<chrono::microseconds>(end
- start).count() << " microseconds" << endl;

    return 0;
}
```

2

```
#include <bits/stdc++.h>
#include <omp.h>

using namespace std;
class Graph
{
public:
    int vertices;
    vector<vector<int>> adj;
    vector<bool> visited;
    Graph(int v) : vertices(v), adj(v), visited(v, false) {}
    void addEgde(int v, int w)
    {
        adj[v].push_back(w);
        adj[w].push_back(w);
    }

    void initialize(){
        for(int i=0;i<vertices;i++){
            visited[i]=false;
        }
    }

    void dfs(int i){
        stack<int>s;
        s.push(i);
        visited[i]=true;

        while(!s.empty()){
            int current=s.top();
            cout<<current<<" ";
            s.pop();

            for(int j=0;j<adj[current].size();j++){
                int neighbours=adj[current][j];
                if(!visited[neighbours]){
                    s.push(neighbours);
                    visited[neighbours]=true;
                }
            }
        }
    }

   void pdfs(int i){
        stack<int>s;
        s.push(i);
        visited[i]=true;

        while(!s.empty()){
            int current=s.top();
            cout<<current<<" ";
            #pragma omp critical
            s.pop();

             #pragma omp parallel for
            for(int j=0;j<adj[current].size();j++){
                int neighbours=adj[current][j];
                if(!visited[neighbours]){
                    #pragma omp critical
                    s.push(neighbours);
                    visited[neighbours]=true;
                }
            }
        }
    }



    void bfs(int i){
        queue<int>q;
        q.push(i);
        visited[i]=true;

        while(!q.empty()){
            int current=q.front();
            q.pop();
            cout<<current<<" ";

            for(int j=0;j<adj[current].size();j++){
                int neighbour=adj[current][j];
                if(!visited[neighbour])
                {
                    q.push(neighbour);
                    visited[neighbour]=true;
                }
            }
        }
    }

    void pbfs(int i){
        queue<int>q;
        q.push(i);
        visited[i]=true;
```

```
        while(!q.empty()){
            int current=q.front();
            #pragma omp critical
            q.pop();
            cout<<current<<" ";

            #pragma omp parallel for
            for(int j=0;j<adj[current].size();j++){
                int neighbour=adj[current][j];
                if(!visited[neighbour])
                {
                    #pragma omp critical
                    q.push(neighbour);
                    visited[neighbour]=true;
                }
            }
        }
    }

};


int main(){

    int n,e;
    cout<<"enter number of edges"<<endl;
    cin>>n;
    Graph g(n);
    cout<<"enter number of edges"<<endl;
    cin>>e;
    for(int i=0;i<e;i++){
        int x,y;
           cout<<"enter edge"<<endl;
           cin>>x>>y;
           g.addEgde(x,y);

    }

    cout<<"dfs"<<endl;
    int start_time = omp_get_wtime();
    g.dfs(0);
    int end_time = omp_get_wtime();
    cout << "Dfs" << end_time - start_time << " seconds\n";
    g.initialize();
    cout<<endl;
    start_time = omp_get_wtime();
    cout<<"pdfs"<<endl;
    g.pdfs(0);
```

```
      end_time = omp_get_wtime();
    cout << "PDfs" << end_time - start_time << " seconds\n";

    g.initialize();
    cout<<endl;
    cout<<"bfs"<<endl;
    start_time = omp_get_wtime();
    g.bfs(0);
     end_time = omp_get_wtime();
    cout << "Bfs" << end_time - start_time << " seconds\n";

    g.initialize();
    cout<<endl;
    cout<<"pbfs"<<endl;
    start_time = omp_get_wtime();
    g.pbfs(0);
      end_time = omp_get_wtime();
    cout << "PBfs" << end_time - start_time << " seconds\n";

    g.initialize();
    cout<<endl;
    return 0;
}
```