

```
1 #ifndef LAB_7_GRAPH_H
2 #define LAB_7_GRAPH_H
3
4 #include <iostream>
5 #include <vector>
6 #include <list>
7 using std::ostream;
8 using std::vector;
9 using std::list;
10
11 class Graph {
12 public:
13     explicit Graph(int V) : v(V), e(0), adjlist(V){ }
14     int V() const;
15
16     int E() const;
17
18     void addEdge(int v, int w);
19
20     const list<int>& adj(int v) const;
21
22     friend ostream &operator<<(ostream &os, const Graph &
graph);
23
24
25 private:
26     vector<list<int>> adjlist;
27     int v;
28     int e;
29 };
30
31
32 #endif //LAB_7_GRAPH_H
33
```

```

1  #include <iostream>
2  #include <limits>
3  #include <sstream>
4  #include <deque>
5  #include <chrono>
6  #include <queue>
7  #include <random>
8  #include "Graph.h"
9
10 using namespace std;
11
12 bool get_line(const string& prompt, string& userinput){
13     cout << prompt;
14     getline(cin, userinput);
15     return !userinput.empty();
16 }
17
18 void bfs(const Graph& G, int source){
19     vector<int> distTo(G.V(), std::numeric_limits<int>::
max());
20     deque<int> edgeTo(G.V(), -1);
21     distTo[source] = 0;
22     edgeTo[source] = 0;
23     queue<int> q;
24     q.push(source);
25     while(!q.empty()){
26         int v = q.front();
27         q.pop();
28         for(int w : G.adj(v)){
29             if(edgeTo[w] == -1){
30                 edgeTo[w] = v;
31                 distTo[w] = distTo[v] + 1;
32                 q.push(w);
33             }
34         }
35     }
36     for(int v = 0; v < G.V(); v++){
37         if(distTo[v] != std::numeric_limits<int>::max()){
38             cout << "Shortest Path cost from: "<< source
<< " to " << v << " is " << distTo[v] << endl;
39             vector<int> path;
40             for(int e = v; e != source; e = edgeTo[e]){
41                 path.push_back(e);
42             }
43             path.push_back(source);
44             for(int i = path.size() - 1; i >= 1; i--){
45                 cout << path[i] << "->";
46             }
47             cout << path[0] << endl;
48         } else {

```

```

49         cout << source << " to " << v << " unreachable
    " << endl;
50     }
51 }
52 }
53
54 enum COLORS{GRAY = 0, RED = 1, BLUE = 2};
55
56 bool is_bipartite(const Graph& G, int source, vector<
COLORS>& colors){
57     colors[source] = BLUE;
58     queue<int> q;
59     q.push(source);
60     while(!q.empty()){
61         int v = q.front();
62         q.pop();
63         for(int w : G.adj(v)){
64             if(colors[w] == GRAY){
65                 colors[w] = (colors[v] == RED) ? BLUE :
RED;
66                 q.push(w);
67             } else if(colors[w] == colors[v]){
68                 cout << "not bipartite" << endl;
69                 return false;
70             }
71         }
72     }
73     return true;
74 }
75
76 void explore(const Graph& G){
77     vector<COLORS > vertex_color(G.V(), GRAY);
78     for(int v = 0; v < G.V(); v++){
79         if(vertex_color[v] == GRAY && !is_bipartite(G, v,
vertex_color)){
80             break;
81         }
82     }
83     vector<string> color_decoded = {"gray", "red", "blue"}
;
84     for(int v = 0; v < G.V(); v++){
85         cout << v << " color: " << color_decoded[
vertex_color[v]] << endl;
86     }
87 }
88
89 Graph generate_graph(int V, int E){
90     vector<pair<int, int>> all_subsets;
91     for(int i = 0; i < V; i++){
92         for(int j = i + 1; j < V; ++j){

```

```

93         all_subsets.push_back({i, j});
94     }
95 }
96 vector<pair<int, int>> subset;
97 for(int i = 0; i < E; i++){
98     subset.push_back(all_subsets[i]);
99 }
100 long seed = chrono::system_clock::now().
time_since_epoch().count();
101 mt19937 gen(seed);
102 uniform_int_distribution<int>
uniform_int_distribution(0, E);
103 for(int i = E; i < all_subsets.size(); i++){
104     int random_idx = uniform_int_distribution(gen);
105     if(random_idx < E) {
106         subset[random_idx] = all_subsets[i];
107     }
108 }
109 Graph G(V);
110 for(const auto& p : subset){
111     G.addEdge(p.first, p.second);
112 }
113 return G;
114 }
115
116 int main() {
117     string userInput;
118     while(get_line("(part a) enter number of vertices
followed by number of edges separated by a space: ",
userInput)){
119         stringstream ss(userInput);
120         int V, E;
121         ss >> V >> E;
122         Graph G = generate_graph(V, E);
123         cout << G << endl;
124         get_line("enter starting vertex of bfs: ",
userInput);
125         ss = stringstream(userInput);
126         int source;
127         ss >> source;
128         bfs(G, source);
129     } while(get_line("(part b) enter number of vertices
followed by number of edges separated by a space: ",
userInput)){
130         stringstream ss(userInput);
131         int V, E;
132         ss >> V >> E;
133         Graph G = generate_graph(V, E);
134         cout << G << endl;
135         explore(G);

```

```
136     }  
137 }
```

```
1 #include "Graph.h"
2
3 int Graph::V() const {
4     return v;
5 }
6
7 ostream &operator<<(ostream &os, const Graph &graph) {
8     os << "Vertices: " << graph.V() << " edges: " << graph
9     .E() << std::endl;
10    for(int v = 0; v < graph.V(); v++){
11        os << v << " : {";
12        for(int w : graph.adj(v)){
13            os << w << " ";
14        }
15        os << "}" << std::endl;
16    }
17    os << std::endl;
18    return os;
19 }
20 int Graph::E() const {
21     return e;
22 }
23
24 void Graph::addEdge(int v, int w){
25     adjlist[v].push_back(w);
26     adjlist[w].push_back(v);
27     e++;
28 }
29
30 const list<int>& Graph::adj(int v) const {
31     return adjlist[v];
32 }
33
```

```
1 cmake_minimum_required(VERSION 3.12)
2 project(lab_7)
3
4 set(CMAKE_CXX_STANDARD 14)
5
6 add_executable(lab_7 main.cpp Graph.cpp Graph.h)
```