**Лабораторная работа №9**

**Поиск расстояний в графе**

**Задание 1**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.
2. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При  реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.

**3.** Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

**Задание 2**

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.
2. Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.
3. Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

Ответ на задание:

#include <iostream>

#include <vector>

#include <queue>

#include <cstdlib>

#include <ctime>

#include <chrono>

#include <windows.h>

using namespace std;

class Graph {

public:

int vertices;

vector<vector<int>> adjacencyMatrix;

vector<vector<int>> adjacencyList;

Graph(int v) : vertices(v) {

// Инициализация матрицы смежности

adjacencyMatrix.resize(vertices, vector<int>(vertices, 0));

// Инициализация списка смежности

adjacencyList.resize(vertices);

}

void addEdge(int u, int v) {

// Добавление ребра в матрицу смежности

adjacencyMatrix[u][v] = 1;

adjacencyMatrix[v][u] = 1;

// Добавление ребра в список смежности

adjacencyList[u].push\_back(v);

adjacencyList[v].push\_back(u);

}

void printMatrix() {

setlocale(LC\_ALL, "rus");

cout << "матрица смежности:\n";

for (int i = 0; i < vertices; ++i) {

for (int j = 0; j < vertices; ++j) {

cout << adjacencyMatrix[i][j] << " ";

}

cout << endl;

}

}

void printList() {

setlocale(LC\_ALL, "rus");

cout << "Список смежности:\n";

for (int i = 0; i < vertices; ++i) {

cout << "Вершин " << i << ": ";

for (int neighbor : adjacencyList[i]) {

cout << neighbor << " ";

}

cout << endl;

}

}

void BFS(int startVertex) {

vector<bool> visited(vertices, false);

queue<int> q;

q.push(startVertex);

visited[startVertex] = true;

while (!q.empty()) {

int currentVertex = q.front();

cout << currentVertex << " ";

q.pop();

for (int neighbor : adjacencyList[currentVertex]) {

if (!visited[neighbor]) {

q.push(neighbor);

visited[neighbor] = true;

}

}

}

cout << endl;

}

void DFS(int startVertex) {

vector<bool> visited(vertices, false);

vector<int> distances(vertices, -1);

DFSUtil(startVertex, visited, distances, 0);

cout << endl;

}

private:

void DFSUtil(int currentVertex, vector<bool>& visited, vector<int>& distances, int distance) {

setlocale(LC\_ALL, "rus");

cout << "Вершин " << currentVertex << ": Расстояние = " << distance << endl;

visited[currentVertex] = true;

distances[currentVertex] = distance;

for (int neighbor : adjacencyList[currentVertex]) {

if (!visited[neighbor]) {

DFSUtil(neighbor, visited, distances, distance + 1);

}

}

}

};

int main() {

setlocale(LC\_ALL, "rus");

srand(time(0));

int vertices = 5; // Задайте количество вершин графа

Graph G(vertices);

// Генерация случайного графа

for (int i = 0; i < vertices - 1; ++i) {

for (int j = i + 1; j < vertices; ++j) {

if (rand() % 2 == 1) {

G.addEdge(i, j);

}

}

}

// Вывод матрицы смежности

G.printMatrix();

// Вывод списка смежности

G.printList();

// Измерение времени выполнения BFS

auto startBFS = chrono::steady\_clock::now();

G.BFS(0);

auto endBFS = chrono::steady\_clock::now();

cout << "времени выполнения BFS: " << chrono::duration\_cast<chrono::microseconds>(endBFS - startBFS).count() << " микросекунды\n";

// Измерение времени выполнения DFS

auto startDFS = chrono::steady\_clock::now();

G.DFS(0);

auto endDFS = chrono::steady\_clock::now();

cout << "Время выполнения DFS: " << chrono::duration\_cast<chrono::microseconds>(endDFS - startDFS).count() << " микросекунды\n";

return 0;

};

