**Министерство науки и высшего образования Российской Федерации**

**Пензенский государственный университет**

**Кафедра «Вычислительная техника»**

**Отчет**

по лабораторной работе №9

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

Выполнили студ. группы 24ВВВ4:

Агуреев А.П.

Невежин И.С.

Приняли:

к.э.н. доцент Акифьев И.В.\_\_\_\_

к.т.н доцент Юрова О.В.\_\_\_\_

Пенза 2025

**Цель -** изучить принцип работы алгоритма поиска расстояний в графе, используя, изученные ранее, процедуры обхода графа в ширину и глубину.

**Лабораторные задания:**

### **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.

**Результат работы программы:**

**Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.**

**Сравнение скорости работы на графов разных порядков:**

**Вывод –** изучили принципы работы алгоритма поиска расстояний в графе, используя, изученные ранее, процедуры обхода графа в ширину и глубину.

**Листинг:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <iostream>

#include <vector>

#include <time.h>

#include <queue>

using namespace std;

struct node {

int index;

struct node\* next\_node;

};

struct Graph {

vector<node\*>vertexes;

};

void BFS\_list(Graph\* G, int start);

void BFS(vector<vector<int>>& M, int start);

void print\_G(Graph\* G);

void add\_el(struct Graph\* G, int vertex, int new\_index);

void DFS\_recurs\_matrix(vector<vector<int>>& M, vector<int>& visited, int count, int depth);

void DFS\_recurs\_list(struct Graph\* G, vector<int>& visited, int count, int depth);

int main()

{

srand(time(0));

clock\_t start, end;

setlocale(LC\_ALL, "Russian");

Graph G;

int sizeM = 0, number = 0;

cout << "Введите количество вершин у графа:\n";

cin >> sizeM;

G.vertexes.resize(sizeM, nullptr);

cout << "Матрица смежности:\n";

vector <vector<int>> M(sizeM, vector<int>(sizeM, 0));

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

for (int j = i; j < sizeM; j++) {

if (i == j) {

M[i][j] = 0;

}

else {

M[i][j] = rand() % 2;

M[j][i] = M[i][j];

}

}

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

printf("%3d ", M[i][j]);

if (M[i][j]) {

if (i == j) {

continue;

}

add\_el(&G, i, j);

}

}

cout << '\n';

}

cout << "Список смежности:\n";

print\_G(&G);

cout << "Введите номер вершины:\n";

cin >> number;

number--;

cout << "Поиск расстояний в графе через обход в ширину:\n";

cout << "Для матрицы:\n";

start = clock();

BFS(M, number);

end = clock();

double time\_spend = (double)(end - start) / CLOCKS\_PER\_SEC;

cout << "Время работы: " << time\_spend << "\n";

cout << "Для списка смежности:\n";

start = clock();

BFS\_list(&G, number);

end = clock();

time\_spend = (double)(end - start) / CLOCKS\_PER\_SEC;

cout << "\nВремя работы: " << time\_spend << "\n\n";

cout << "Поиск расстояний в графе используя обход в глубину:\n";

cout << "Для матрицы:\n";

vector<int> visited(sizeM, -1);

start = clock();

DFS\_recurs\_matrix(M, visited, number, 0);

end = clock();

cout << "Расстояния от вершины " << number + 1 << ":\n";

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

cout << "До " << i + 1 << ": " << visited[i] << '\n';

}

time\_spend = (double)(end - start) / CLOCKS\_PER\_SEC;

cout << "Время работы: " << time\_spend << '\n';

cout << "Для списка:\n";

visited.assign(sizeM, -1);

start = clock();

DFS\_recurs\_list(&G, visited, number, 0);

end = clock();

cout << "Расстояния от вершины " << number + 1 << ":\n";

for (int i = 0; i < G.vertexes.size(); i++) {

cout << "До " << i + 1 << ": " << visited[i] << '\n';

}

time\_spend = (double)(end - start) / CLOCKS\_PER\_SEC;

cout << "\nВремя работы: " << time\_spend << '\n';

}

void BFS(vector<vector<int>>& M, int start) {

vector<int> visited(M.size(), -1);

queue<int> q;

q.push(start);

visited[start] = 0;

while (!q.empty()) {

int current = q.front();

q.pop();

for (int i = 0; i < M.size(); i++) {

if (M[current][i] == 1 && visited[i] == -1) {

visited[i] = visited[current] + 1;

q.push(i);

}

}

}

cout << "Расстояние от вершины " << start + 1 << ":\n";

for (int i = 0; i < M.size(); i++)

cout << "До " << i + 1 << ": " << visited[i] << '\n';

}

void BFS\_list(Graph\* G, int start) {

vector<int> visited(G->vertexes.size(), -1);

queue<int> q;

q.push(start);

visited[start] = 0;

while (!q.empty()) {

int current = q.front();

q.pop();

node\* tmp = G->vertexes[current];

while (tmp != nullptr) {

if (visited[tmp->index] == -1) {

visited[tmp->index] = visited[current] + 1;

q.push(tmp->index);

}

tmp = tmp->next\_node;

}

}

cout << "Расстояние от вершины " << start + 1 << ":\n";

for (int i = 0; i < G->vertexes.size(); i++) {

cout << "До " << i + 1 << ": " << visited[i] << '\n';

}

}

void DFS\_recurs\_matrix(vector<vector<int>>& M, vector<int>& visited, int current, int depth) {

visited[current] = depth;

for (int i = 0; i < M.size(); i++) {

if (M[current][i] && visited[i] == -1) {

DFS\_recurs\_matrix(M, visited, i, depth+1);

}

}

}

void DFS\_recurs\_list(struct Graph\* G, vector<int>& visited, int current, int depth) {

visited[current] = depth;

node\* tmp = G->vertexes[current];

while (tmp != nullptr) {

if (visited[tmp->index] == -1) {

DFS\_recurs\_list(G, visited, tmp->index, depth + 1);

}

tmp = tmp->next\_node;

}

}

void add\_el(struct Graph\* G, int vertex, int new\_index) {

node\* p = (node\*)malloc(sizeof(node));

p->index = new\_index;

if (G->vertexes[vertex] == nullptr) {

G->vertexes[vertex] = p;

p->next\_node = nullptr;

}

else {

node\* tmp = G->vertexes[vertex];

while (tmp->next\_node != nullptr) {

tmp = tmp->next\_node;

}

tmp->next\_node = p;

p->next\_node = nullptr;

}

}

void print\_G(Graph\* G) {

for (int i = 0; i < G->vertexes.size(); i++) {

node\* tmp = G->vertexes[i];

cout << "Вершина " << i + 1 << ": ";

while (tmp != nullptr) {

cout << tmp->index + 1 << " ";

tmp = tmp->next\_node;

}

cout << '\n';

}

}