Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

**Отчет**

по лабораторной работе № 10

по курсу “Логика и основы алгоритмизации в инженерных задачах”

на тему “ Поиск расстояний во взвешенном графе”

Выполнили студенты группы 22ВВС1:

Разин Д.С

Беккаревич К.А

Приняли:

Акифьев И. В.

Юрова О.В

Пенза 2023

**Лабораторное задание:**

**Задание 1**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного взвешенного 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.**\* Сгенерируйте (используя генератор случайных чисел) матрицу смежности для ориентированного взвешенного графа *G*. Выведите матрицу на экран и осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием.

**Задание 2**

1. Для каждого из вариантов сгенерированных графов (ориентированного и не ориентированного) определите радиус и диаметр.
2. Определите подмножества периферийных и центральных вершин.

**Задание 3\***

1. Модернизируйте программу так, чтобы получить возможность запуска

программы с параметрами командной строки (см. описание ниже). В

качестве параметра должны указываться тип графа (взвешенный или нет) и

наличие ориентации его ребер (есть ориентация или нет).

**Задание №10.1.2**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <queue>

#include <clocale>

#define MAX\_NODES 400

// Функция для генерации случайной матрицы смежности

void generateRandomGraph(int graph[MAX\_NODES][MAX\_NODES], int n) {

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

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

if (i == j) {

graph[i][j] = 0; // Нет петель

}

else {

int random\_value = rand() % 2; // Генерируем 0 или 1

graph[i][j] = random\_value;

graph[j][i] = random\_value; // Граф неориентированный

}

}

}

}

// Функция для поиска расстояний в графе с использованием BFS

void bfs(int graph[MAX\_NODES][MAX\_NODES], int n, int startNode) {

bool visited[MAX\_NODES] = { false };

int distances[MAX\_NODES] = { 0 };

std::queue<int> q;

q.push(startNode);

visited[startNode] = true;

while (!q.empty()) {

int currentNode = q.front();

q.pop();

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

if (graph[currentNode][i] && !visited[i]) {

q.push(i);

visited[i] = true;

distances[i] = distances[currentNode] + 1;

}

}

}

// Выводим расстояния от startNode до всех остальных узлов

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

if (distances[i] == 0) {

printf("До вершины %d: Недостижима\n", i);

}

else {

printf("До вершины %d: %d\n", i, distances[i]);

}

}

}

// Функция для определения эксцентриситета вершины

int findEccentricity(int graph[MAX\_NODES][MAX\_NODES], int n, int vertex) {

bool visited[MAX\_NODES] = { false };

int distances[MAX\_NODES] = { 0 };

std::queue<int> q;

q.push(vertex);

visited[vertex] = true;

while (!q.empty()) {

int currentNode = q.front();

q.pop();

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

if (graph[currentNode][i] && !visited[i]) {

q.push(i);

visited[i] = true;

distances[i] = distances[currentNode] + 1;

}

}

}

// Находим максимальное расстояние (эксцентриситет)

int eccentricity = 0;

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

if (distances[i] > eccentricity) {

eccentricity = distances[i];

}

}

return eccentricity;

}

// Функция для определения радиуса графа

int findRadius(int graph[MAX\_NODES][MAX\_NODES], int n) {

int minEccentricity = INT\_MAX;

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

int eccentricity = findEccentricity(graph, n, i);

if (eccentricity < minEccentricity) {

minEccentricity = eccentricity;

}

}

return minEccentricity;

}

// Функция для определения диаметра графа

int findDiameter(int graph[MAX\_NODES][MAX\_NODES], int n) {

int maxEccentricity = 0;

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

int eccentricity = findEccentricity(graph, n, i);

if (eccentricity > maxEccentricity) {

maxEccentricity = eccentricity;

}

}

return maxEccentricity;

}

// Функция для определения центральных вершин

void findCentralVertices(int graph[MAX\_NODES][MAX\_NODES], int n, int radius) {

printf("Центральные вершины (эксцентриситет равен радиусу графа):\n");

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

int eccentricity = findEccentricity(graph, n, i);

if (eccentricity == radius) {

printf("%d\n", i);

}

}

}

// Функция для определения периферийных вершин

void findPeripheralVertices(int graph[MAX\_NODES][MAX\_NODES], int n, int diameter) {

printf("Периферийные вершины (эксцентриситет равен диаметру гра-фа):\n");

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

int eccentricity = findEccentricity(graph, n, i);

if (eccentricity == diameter) {

printf("%d\n", i);

}

}

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(NULL));

int n;

printf("Введите количество вершин в графе: ");

scanf("%d", &n);

while (n == 0) {

printf("Ошибка, Введите занова количество вершин в графе: ");

scanf("%d", &n);

}

int graph[MAX\_NODES][MAX\_NODES] = { 0 };

generateRandomGraph(graph, n);

// Выводим матрицу смежности

printf("Матрица смежности графа:\n");

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

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

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

}

printf("\n");

}

int startNode;

printf("Введите начальный узел для поиска расстояний(от 0 до %d): ", n - 1);

scanf("%d", &startNode);

int distances[MAX\_NODES];

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

distances[i] = INT\_MAX;

}

bfs(graph, n, startNode);

// Определение радиуса и диаметра

int radius = findRadius(graph, n);

int diameter = findDiameter(graph, n);

printf("Радиус графа: %d\n", radius);

printf("Диаметр графа: %d\n", diameter);

// Определение центральных и периферийных вершин

findCentralVertices(graph, n, radius);

findPeripheralVertices(graph, n, diameter);

getchar();

getchar();

return 0;

}

}

**Задание №10.1.3**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <clocale>

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

// Функция для поиска минимального расстояния

int minDistance(int dist[], int sptSet[], int V) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (sptSet[v] == 0 && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

// Функция для вывода матрицы смежности

void printMatrix(int\*\* graph, int V) {

printf("Матрица смежности:\n");

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

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

printf("%d\t", graph[i][j]);

}

printf("\n");

}

}

// Функция для вывода результатов поиска расстояний

void printSolution(int dist[], int V) {

printf("Вершина \t Расстояние от начальной вершины\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

}

// Функция для реализации поиска расстояний в графе

void dijkstra(int\*\* graph, int src, int V, int\* dist) {

int\* sptSet = (int\*)malloc(V \* sizeof(int)); // Массив для отслеживания посещенных вершин

// Инициализация массива расстояний и массива посещенных вершин

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

// Расстояние от начальной вершины до неё самой равно 0

dist[src] = 0;

// Поиск кратчайшего пути для всех вершин

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

// Помечаем выбранную вершину как посещенную

sptSet[u] = 1;

// Обновляем значения расстояний для соседей выбранной вершины

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

free(sptSet);

}

// Функция для определения радиуса графа

int findRadius(int\*\* graph, int V) {

int minEccentricity = INT\_MAX;

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance < minEccentricity) {

minEccentricity = maxDistance;

}

free(dist);

}

return minEccentricity;

}

// Функция для определения диаметра графа

int findDiameter(int\*\* graph, int V) {

int maxEccentricity = 0;

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance > maxEccentricity) {

maxEccentricity = maxDistance;

}

free(dist);

}

return maxEccentricity;

}

// Функция для определения центральных вершин

void findCentralVertices(int\*\* graph, int V, int radius) {

printf("Центральные вершины (эксцентриситет равен радиусу графа):\n");

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance == radius) {

printf("%d\n", i);

}

free(dist);

}

}

// Функция для определения периферийных вершин

void findPeripheralVertices(int\*\* graph, int V, int diameter) {

printf("Периферийные вершины (эксцентриситет равен диаметру графа):\n");

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance == diameter) {

printf("%d\n", i);

}

free(dist);

}

}

int main() {

setlocale(LC\_ALL, "Russian");

int V; // Количество вершин

printf("Введите количество вершин в графе: ");

scanf("%d", &V);

// Создание графа и автоматическое заполнение матрицы смежности

int\*\* graph = (int\*\*)malloc(V \* sizeof(int\*));

// Инициализация генератора случайных чисел

srand(time(NULL));

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

graph[i] = (int\*)malloc(V \* sizeof(int));

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

if (i == j) {

graph[i][j] = 0; // Расстояние от вершины до самой себя

}

else {

// Заполняем случайными весами от 1 до 10

graph[i][j] = rand() % 10 + 1;

}

}

}

printMatrix(graph, V);

// Создание матрицы для хранения расстояний

int\* dist = (int\*)malloc(V \* sizeof(int));

// Выбор начальной вершины для поиска расстояний

int source;

printf("Введите начальную вершину для поиска расстояний: ");

scanf("%d", &source);

// Выполнение поиска расстояний для одной из вершин (можно выбрать любую)

dijkstra(graph, source, V, dist);

// Вывод результатов

printSolution(dist, V);

// Определение радиуса и диаметра

int radius = findRadius(graph, V);

int diameter = findDiameter(graph, V);

printf("Радиус графа: %d\n", radius);

printf("Диаметр графа: %d\n", diameter);

// Определение центральных и периферийных вершин

findCentralVertices(graph, V, radius);

findPeripheralVertices(graph, V, diameter);

// Освобождение выделенной памяти

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

free(graph[i]);

}

free(graph);

free(dist);

return 0;

}

Задание 10.1.4

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <clocale>

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#include <string>

// Функция для поиска минимального расстояния

int minDistance(int dist[], int sptSet[], int V) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (sptSet[v] == 0 && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

// Функция для вывода матрицы смежности

void printMatrix(int\*\* graph, int V) {

printf("Матрица смежности:\n");

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

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

printf("%d\t", graph[i][j]);

}

printf("\n");

}

}

// Функция для вывода результатов поиска расстояний

void printSolution(int dist[], int V) {

printf("Вершина \t Расстояние от начальной вершины\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

}

// Функция для реализации поиска расстояний в графе

void dijkstra(int\*\* graph, int src, int V, int\* dist) {

int\* sptSet = (int\*)malloc(V \* sizeof(int)); // Массив для отслеживания посещенных вершин

// Инициализация массива расстояний и массива посещенных вершин

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

// Расстояние от начальной вершины до неё самой равно 0

dist[src] = 0;

// Поиск кратчайшего пути для всех вершин

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

// Помечаем выбранную вершину как посещенную

sptSet[u] = 1;

// Обновляем значения расстояний для соседей выбранной вершины

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

free(sptSet);

}

// Функция для определения радиуса графа

int findRadius(int\*\* graph, int V) {

int minEccentricity = INT\_MAX;

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance < minEccentricity) {

minEccentricity = maxDistance;

}

free(dist);

}

return minEccentricity;

}

// Функция для определения диаметра графа

int findDiameter(int\*\* graph, int V) {

int maxEccentricity = 0;

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance > maxEccentricity) {

maxEccentricity = maxDistance;

}

free(dist);

}

return maxEccentricity;

}

// Функция для определения центральных вершин

void findCentralVertices(int\*\* graph, int V, int radius) {

printf("Центральные вершины (эксцентриситет равен радиусу графа):\n");

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance == radius) {

printf("%d\n", i);

}

free(dist);

}

}

// Функция для определения периферийных вершин

void findPeripheralVertices(int\*\* graph, int V, int diameter) {

printf("Периферийные вершины (эксцентриситет равен диаметру гра-фа):\n");

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

int\* dist = (int\*)malloc(V \* sizeof(int));

dijkstra(graph, i, V, dist);

int maxDistance = 0;

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

if (dist[j] != INT\_MAX && dist[j] > maxDistance) {

maxDistance = dist[j];

}

}

if (maxDistance == diameter) {

printf("%d\n", i);

}

free(dist);

}

}

using namespace std;

int main(int argc, char\* argv[]) {

setlocale(LC\_ALL, "Russian");

// Создание графа и автоматическое заполнение матрицы смежности

int\*\* graph = (int\*\*)malloc(std::stoi(argv[1]) \* sizeof(int\*));

// Инициализация генератора случайных чисел

srand(time(NULL));

for (int i = 0; i < std::stoi(argv[1]); i++) {

graph[i] = (int\*)malloc(std::stoi(argv[1]) \* sizeof(int));

for (int j = 0; j < std::stoi(argv[1]); j++) {

if (i == j) {

graph[i][j] = 0; // Расстояние от вершины до самой себя

}

else {

// Заполняем случайными весами от 1 до 10

graph[i][j] = rand() % 10 + 1;

}

}

}

printMatrix(graph, std::stoi(argv[1]));

// Создание матрицы для хранения расстояний

int\* dist = (int\*)malloc(std::stoi(argv[1]) \* sizeof(int));

// Выбор начальной вершины для поиска расстояний

// Выполнение поиска расстояний для одной из вершин (можно выбрать лю-бую)

dijkstra(graph, std::stoi(argv[2]), std::stoi(argv[1]), dist);

// Вывод результатов

printSolution(dist, std::stoi(argv[1]));

// Определение радиуса и диаметра

int radius = findRadius(graph, std::stoi(argv[1]));

int diameter = findDiameter(graph, std::stoi(argv[1]));

printf("Радиус графа: %d\n", radius);

printf("Диаметр графа: %d\n", diameter);

// Определение центральных и периферийных вершин

findCentralVertices(graph, std::stoi(argv[1]), radius);

findPeripheralVertices(graph, std::stoi(argv[1]), diameter);

// Освобождение выделенной памяти

for (int i = 0; i < std::stoi(argv[1]); i++) {

free(graph[i]);

}

free(graph);

free(dist);

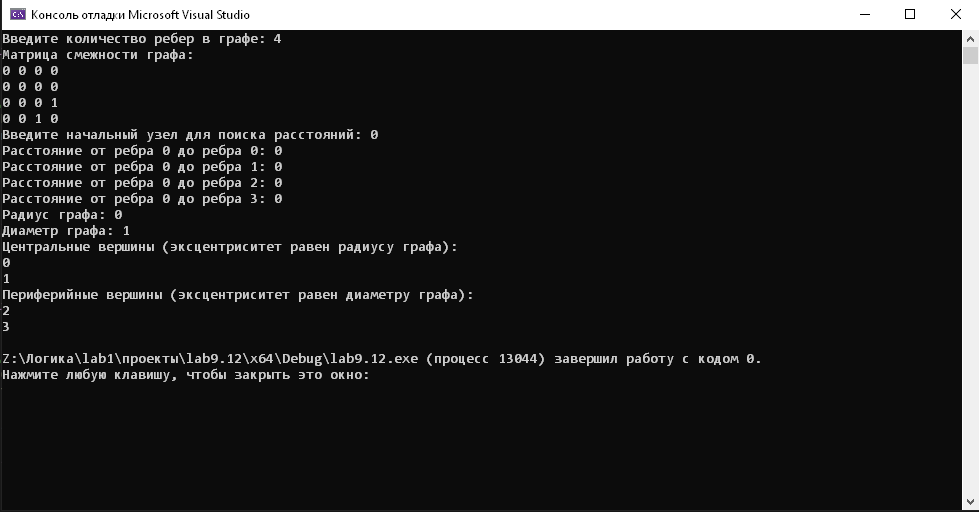
getchar();

getchar();

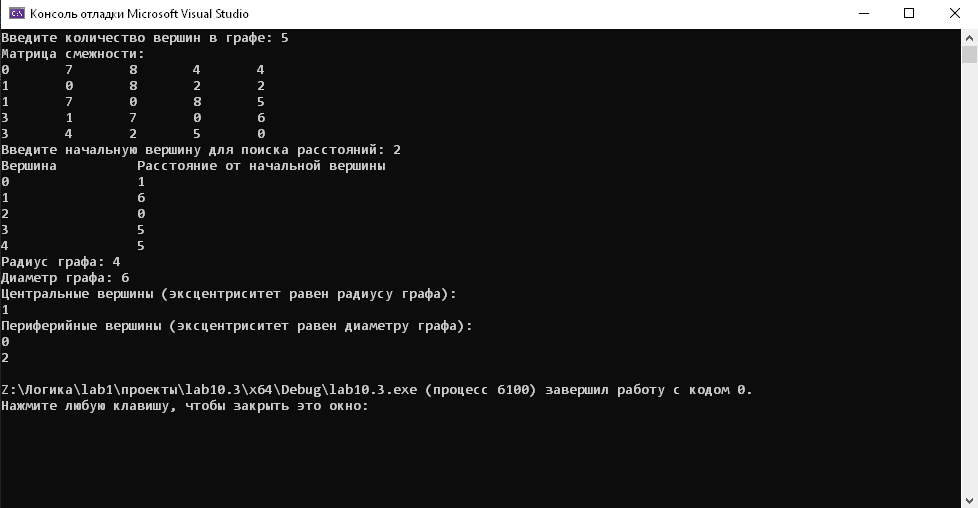
return 0;

}

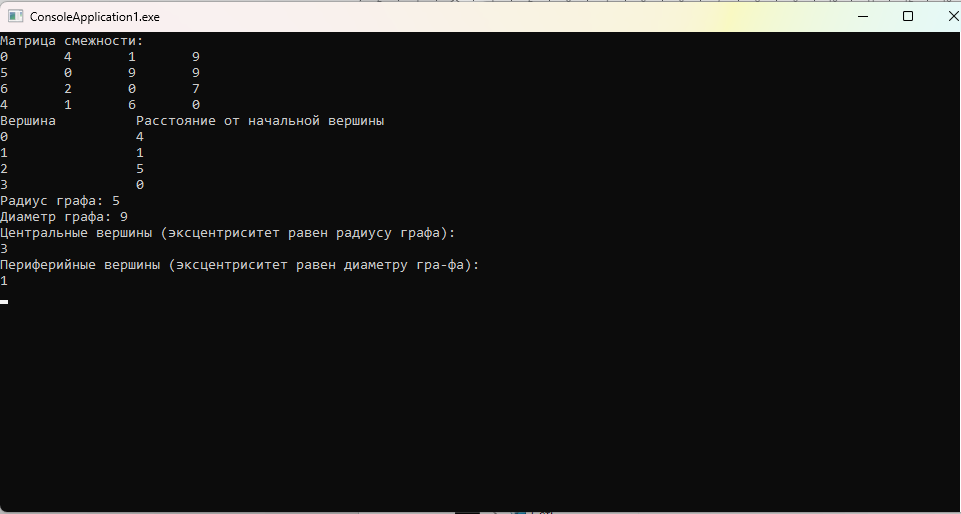
**Результаты работы программ**



**Рисунок 1 - Результат работы программы lab10.1.2**



**Рисунок 2 - Результат работы программы lab10.1.3**



**Рисунок 3 - Результат работы программы lab10.1.4**

**Вывод**

В результате выполнения лабораторной работы были успешно реализованы алгоритмы построения матрицы смежности, усвоены и практически проверены навыки в алгоритме реализации поиска расстояний во взвешенном графе, а также реализован поиск диаметра и радиуса графов, изучили и нашли подмножества периферийных и центральных вершин.