Министерство науки и высшего образования РФ

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 22ВВП2:

Широкова И.Д.

Перкин П.О.

Приняли

Акифьев И.В.

Юрова О.В.

Пенза 2023

**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 неориентированного) 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).

**Листинг**

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <queue>

#include <climits>

using namespace std;

struct Node {

int data;

Node\* next;

};

void printAdjacencyList(Node\*\* adjacencyList, int size) {

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

cout << "Вершина " << i << " смежна с: ";

Node\* current = adjacencyList[i];

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

}

Node\*\* adjacencyMatrixToAdjacencyList(int\*\* matrix, int size) {

Node\*\* adjacencyList = new Node \* [size];

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

adjacencyList[i] = nullptr;

for (int j = size - 1; j >= 0; j--) {

if (matrix[i][j] != 0) {

Node\* newNode = new Node;

newNode->data = j;

newNode->next = adjacencyList[i];

adjacencyList[i] = newNode;

}

}

}

return adjacencyList;

}

void freeAdjacencyList(Node\*\* adjacencyList, int size) {

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

Node\* current = adjacencyList[i];

while (current != nullptr) {

Node\* next = current->next;

delete current;

current = next;

}

}

delete[] adjacencyList;

}

class Node1 {

public:

int data;

int priority;

Node1\* next;

Node1(int d, int p) : data(d), priority(p), next(nullptr) {}

};

class PriorityQueue {

private:

Node1\* front;

public:

PriorityQueue() : front(nullptr) {}

// Вставка элемента с учетом приоритета

void push(int data, int priority) {

Node1\* newNode1 = new Node1(data, priority);

if (!front || priority < front->priority) {

newNode1->next = front;

front = newNode1;

}

else {

Node1\* current = front;

while (current->next && current->next->priority <= priority) {

current = current->next;

}

newNode1->next = current->next;

current->next = newNode1;

}

}

// Извлечение элемента с наивысшим приоритетом

void pop() {

if (front) {

Node1\* temp = front;

front = front->next;

delete temp;

}

}

// Получение значения элемента с наивысшим приоритетом

int top() {

if (front) {

return front->data;

}

throw std::runtime\_error("Queue is empty");

}

// Проверка, пуста ли очередь

bool isEmpty() {

return front == nullptr;

}

};

int\*\* createMatrix(int size) {

int\*\* matrix = new int\*[size];

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

matrix[i] = new int[size];

}

return matrix;

}

void deleteMatrix(int\*\* matrix, int size) {

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

delete[] matrix[i];

}

delete[] matrix;

}

int\*\* generateAdjacencyMatrix(int size) {

int\*\* matrix = new int\*[size];

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

matrix[i] = new int[size];

}

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

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

if (i == j) {

matrix[i][j] = 0;

}

else {

int randomValue = rand() % 5;

if (randomValue == 0) {

matrix[i][j] = 1;

matrix[j][i] = 1;

}

else {

matrix[i][j] = 0;

matrix[j][i] = 0;

}

}

}

}

return matrix;

}

//направленная

int\*\* generateDirectedMatrix(int size) {

int\*\* matrix = new int\*[size];

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

matrix[i] = new int[size];

}

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

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

if (i == j) {

matrix[i][j] = 0;

}

else {

int randomValue = rand() % 5;

if (randomValue == 0) {

matrix[i][j] = 1;

}

else {

matrix[i][j] = 0;

}

}

}

}

return matrix;

}

int summMatrix(int\*\* matrix, int size) {

int summ = 0;

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

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

summ += matrix[i][j];

}

}

return summ;

}

void printMatrix(int\*\* matrix, int size) {

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

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

//cout << matrix[i][j];

printf("%2d", matrix[i][j]);

cout << ", ";

}

cout << endl;

}

}

// Функция для генерации матрицы смежности для неориентированного взвешенного графа

int\*\* generateWeightedAdjacencyMatrix(int size) {

int\*\* matrix = new int\*[size];

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

matrix[i] = new int[size];

}

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

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

if (i == j) {

matrix[i][j] = 0;

}

else {

int rnd = rand() % 10;

// Генерируем 0 с вероятностью 50%

if (rnd < 5) {

matrix[i][j] = 0;

matrix[j][i] = 0;

}

else {

int weight = rand() % 20;

matrix[i][j] = weight;

matrix[j][i] = weight;

}

}

}

}

return matrix;

}

// Функция для генерации матрицы смежности для ориентированного взвешенного графа

int\*\* generateWeightedDirectedAdjacencyMatrix(int size) {

int\*\* matrix = new int\*[size];

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

matrix[i] = new int[size];

}

// Заполняем матрицу случайными весами (в диапазоне от 0 до 9)

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

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

if (i == j) {

matrix[i][j] = 0; // На главной диагонали всегда 0

}

else {

int rnd = rand() % 10;

// Генерируем 0 с вероятностью 70%

if (rnd < 7) {

matrix[i][j] = 0;

}

else {

int weight = rand() % 20;

matrix[i][j] = weight;

}

}

}

}

return matrix;

}

// Функция для поиска кратчайших расстояний с использованием алгоритма Дейкстры

void Dijkstra(int\*\* G, int size, int v, int\* DIST) {

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

DIST[i] = -1; // Инициализируем расстояния как -1 (не посещено)

}

// Создаем приоритетную очередь для обхода вершин

PriorityQueue pq;

// Помещаем стартовую вершину в очередь

pq.push(v, 0);

// Расстояние до самой себя равно 0

DIST[v] = 0;

while (!pq.isEmpty()) {

int currentVertex = pq.top(); // Получаем вершину с наименьшим расстоянием

pq.pop(); // Удаляем вершину из очереди

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

if (G[currentVertex][i] > 0 && (DIST[i] == -1 || DIST[i] > DIST[currentVertex] + G[currentVertex][i])) {

// Если есть ребро между текущей вершиной и i и i еще не посещена

// Или новый путь короче, чем старый

pq.push(i, DIST[currentVertex] + G[currentVertex][i]); // Помещаем вершину в очередь с обновленным расстоянием

DIST[i] = DIST[currentVertex] + G[currentVertex][i]; // Обновляем расстояние

}

}

}

}

int\* findRadiusAndDiameter(int\*\* G, int size, int& radius, int& diameter) {

radius = INT\_MAX;

diameter = 0;

int\* eccentricities = new int[size];

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

int\* distances = new int[size];

Dijkstra(G, size, i, distances);

int maxEccentricity = 0;

int minEccentricity = INT\_MAX;

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

if (distances[j] > maxEccentricity) {

maxEccentricity = distances[j];

}

}

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

if (distances[j] < minEccentricity && distances[j]>0) {

minEccentricity = distances[j];

}

}

eccentricities[i] = maxEccentricity;

if (maxEccentricity < radius && maxEccentricity > 0 ) {

radius = maxEccentricity;

}

if (maxEccentricity > diameter) {

diameter = maxEccentricity;

}

delete[] distances;

}

return eccentricities;

}

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

setlocale(LC\_ALL, "Rus");

srand(time(0));

if (argc != 3) {

cerr << "Usage: " << argv[0] << " <weighted (0 or 1)> <directed (0 or 1)>" << endl;

return 1;

}

int weighted = atoi(argv[1]);

int directed = atoi(argv[2]);

int size;

cout << "\nВведите размер матрицы смежности: ";

cin >> size;

int\*\* adjacencyMatrix;

if (weighted) {

if (directed) {

adjacencyMatrix = generateWeightedDirectedAdjacencyMatrix(size);

}

else {

adjacencyMatrix = generateWeightedAdjacencyMatrix(size);

}

}

else {

if (directed) {

adjacencyMatrix = generateDirectedMatrix(size);

}

else {

adjacencyMatrix = generateAdjacencyMatrix(size);

}

}

printMatrix(adjacencyMatrix, size);

cout << "\nAdjacency List:\n";

Node\*\* adjacencyList = adjacencyMatrixToAdjacencyList(adjacencyMatrix, size);

printAdjacencyList(adjacencyList, size);

int\* DIST = new int[size];

int radius, diameter;

int sum = summMatrix(adjacencyMatrix, size);

cout << "\nПоиск расстояний:\n";

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

Dijkstra(adjacencyMatrix, size, j, DIST);

cout << "Вершина " << j << "\n";

// Вывод расстояний до всех вершин

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

cout << "Расстояние до вершины " << i << " = " << DIST[i] << endl;

}

cout << "\n";

}

int\* eccentricities = findRadiusAndDiameter(adjacencyMatrix, size, radius, diameter);

if (size == 0 || size == 1 || sum == 0) {

cout << "Радиус графа: " << " " << endl;

cout << "Диаметр графа: " << " " << endl;

}

else{

cout << "Радиус графа: " << radius << endl;

cout << "Диаметр графа: " << diameter << endl;

}

if (size == 1 || sum == 0) cout << "Центральные вершины: " << endl; else {

cout << "Центральные вершины: ";

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

if (eccentricities[i] == radius) {

cout << i << " ";

}

}

cout << endl;

}

if (size == 1 || sum == 0) cout << "Перефирийные вершины: " << endl; else {

cout << "Перефирийные вершины: ";

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

if (eccentricities[i] == diameter) {

cout << i << " ";

}

}

cout << endl;

}

delete[] eccentricities;

if (weighted) {

deleteMatrix(adjacencyMatrix, size);

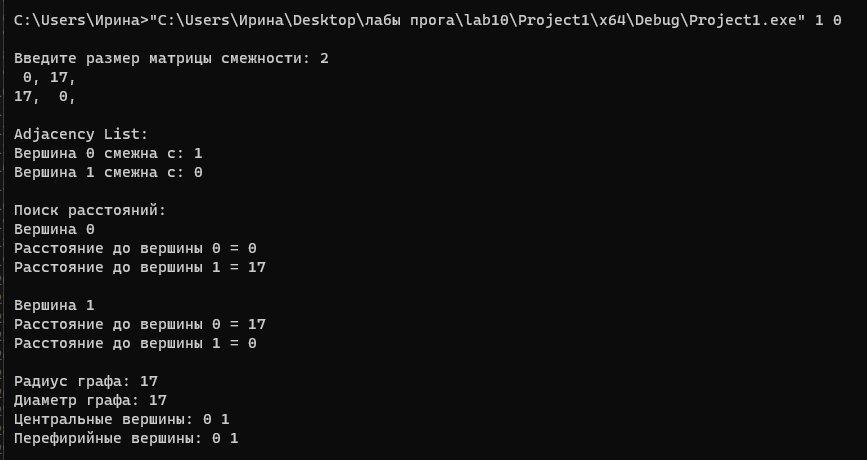
}

delete[] DIST;

return 0;

}

**Результаты работы программы**

****

**Вывод**

В ходе выполнения заданий был успешно проведен поиск расстояний в графах, как неориентированных, так и ориентированных, с весами на ребрах. Задачи определения радиуса, диаметра, а также выделения периферийных и центральных вершин дали более глубокое понимание структуры графов, а модернизация программы для работы с параметрами командной строки повысила ее гибкость и удобство использования.