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:

Бормотов А.А.

Кочетков А.М.

Кузьмин Д.В.

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>**

**#include <string.h>**

**using namespace std;**

**class Node {**

**public:**

**int data;**

**int priority;**

**Node\* next;**

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

**};**

**class PriorityQueue {**

**private:**

**Node\* front;**

**public:**

**PriorityQueue() : front(nullptr) {}**

**void push(int data, int priority) {**

**Node\* newNode = new Node(data, priority);**

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

**newNode->next = front;**

**front = newNode;**

**}**

**else {**

**Node\* current = front;**

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

**current = current->next;**

**}**

**newNode->next = current->next;**

**current->next = newNode;**

**}**

**}**

**void pop() {**

**if (front) {**

**Node\* temp = front;**

**front = front->next;**

**delete temp;**

**}**

**}**

**// get element with topnotch priority**

**int top() {**

**if (front) {**

**return front->data;**

**}**

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

**}**

**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() % 2;**

**if (randomValue == 0) {**

**matrix[i][j] = 1;**

**matrix[j][i] = 1;**

**}**

**else {**

**matrix[i][j] = 0;**

**matrix[j][i] = 0;**

**}**

**}**

**}**

**}**

**return matrix;**

**}**

**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() % 2;**

**if (randomValue == 0) {**

**matrix[i][j] = 1;**

**}**

**else {**

**matrix[i][j] = 0;**

**}**

**}**

**}**

**}**

**return matrix;**

**}**

**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;**

**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];**

**}**

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

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

**if (i == j) {**

**matrix[i][j] = 0;**

**}**

**else {**

**int rnd = rand() % 10;**

**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;**

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

**if (distances[j] > maxEccentricity) {**

**maxEccentricity = distances[j];**

**}**

**}**

**eccentricities[i] = maxEccentricity;**

**if (maxEccentricity < radius) {**

**radius = maxEccentricity;**

**}**

**if (maxEccentricity > diameter) {**

**diameter = maxEccentricity;**

**}**

**delete[] distances;**

**}**

**return eccentricities;**

**}**

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

**setlocale(LC\_ALL, "Rus");**

**srand(time(0));**

**int weighted, directed;**

**//if (argc != 3) {**

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

**// return 1;**

**//}**

**if (argc > 1) {**

**for (int i = 0; i < argc; i++) {**

**if (strcmp(argv[i], "-weighted") == 0) {**

**weighted = atoi(argv[i + 1]);**

**}**

**if (strcmp(argv[i], "-directed") == 0) {**

**directed = atoi(argv[i + 1]);**

**}**

**}**

**}**

**else {**

**cout << "Not enough arguments or too many args" << endl;**

**cout << "Type: 'program\_name'.exe -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);**

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

**int radius, diameter;**

**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);**

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

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

**cout << "Центральные вершины: ";**

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

**if (eccentricities[i] == radius) {**

**cout << i << " ";**

**}**

**}**

**cout << endl;**

**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);**

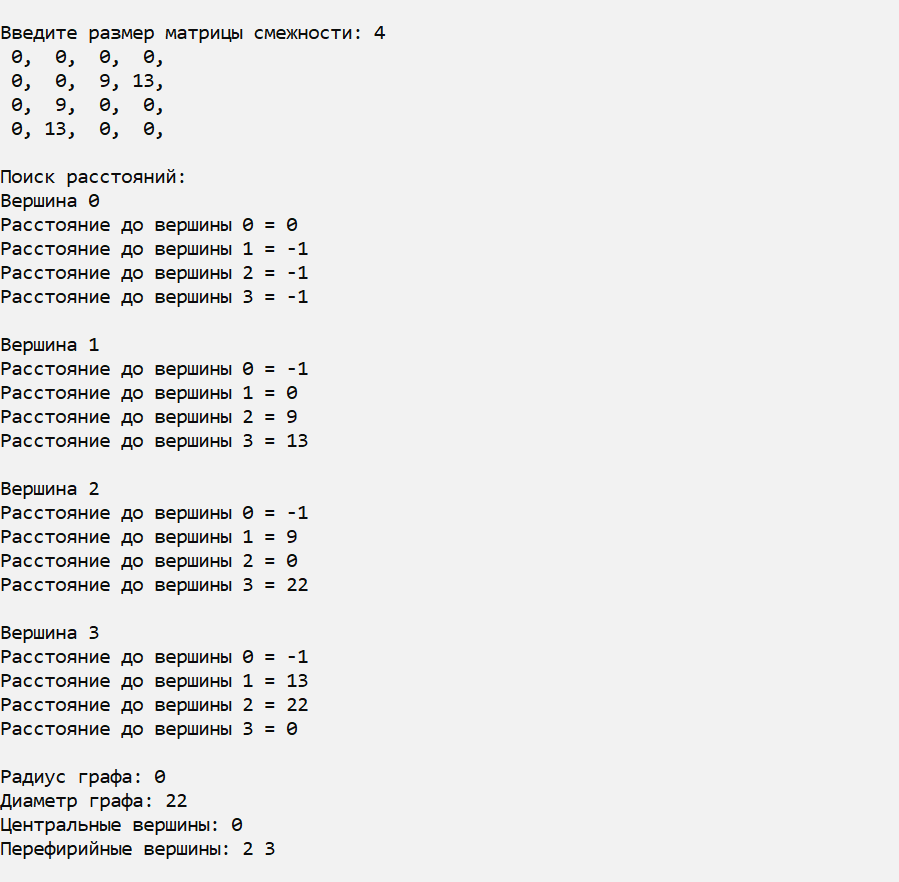
**}**

**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.