Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №10

по теме «Поиск расстояний во взвешенном графе»

Выполнили:

Студенты группы 21ВВ2

Назаров Е.А.

Макаров И.С.

Принял:

Митрохин М.А.

Юрова О.В.

Пенза, 2022

**Цель работы:**

Решение задач, использующих графовые модели представления, изучение различных методов решения и анализа.Поиск кратчайшего пути во взвешенном графе, алгоритм дейкстры.

**Лабораторные работы:**

**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.

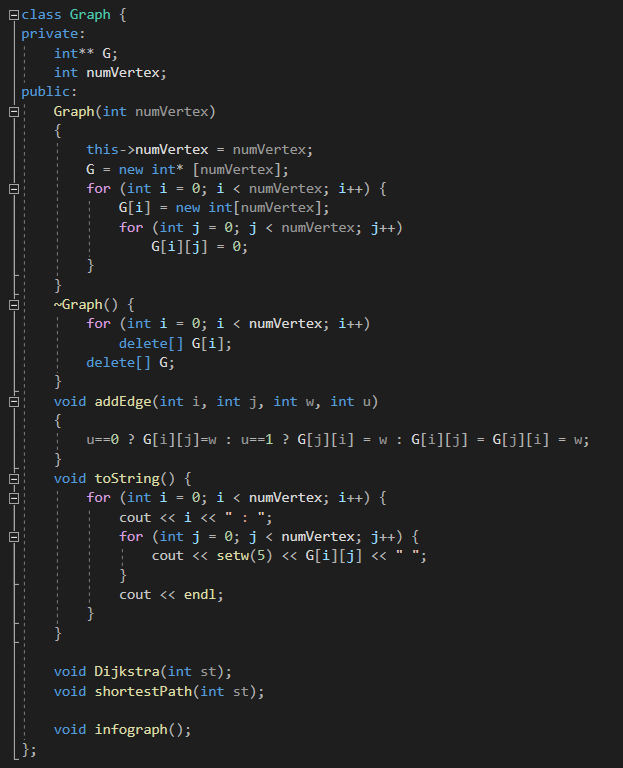
**Задание 2**

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).

**Пояснительный текст к программе:**

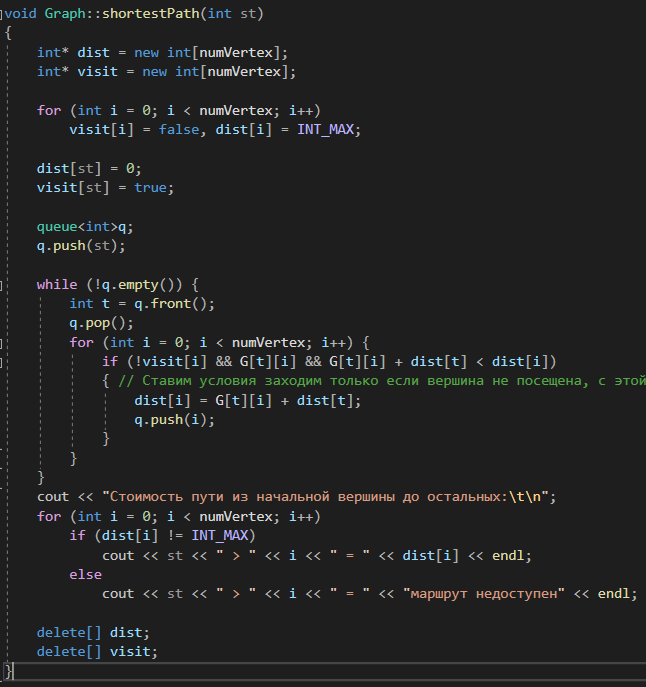


Класс содержащий в себе информацию о взвешенном графе. В конструктор поступает количество вершин, после чего выделяется динамическая память для всех нужных элементов и массивов. Деструктор в свою очередь освобождает эту память при завершении программы.

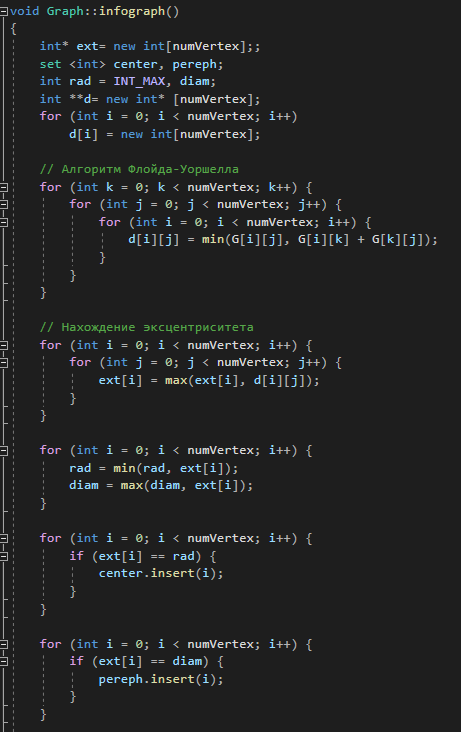
Функции вывода матрицы смежности на экран(toString) и добавления связей между вершинами(addEdge). Функция добавления ребра между вершинами принимает в себя 4 значения: первые два элемента - номера вершин, третий элемент – размер ребра (его масса), четвертый элемент – принцип добавления связей между вершинами (используются для ориентированного графа)



Реализация алгоритма дейкстры. Пусть расстояние от начальной вершины start до вершины i хранится в массиве dist[i]. Начальные значения dist[start]=0, dist[i]=INF для всех остальных вершин i. То есть в самом начале алгоритму известен путь из вершины start до вершины start длины 0, а до остальных вершин кратчайшие пути неизвестны. Между тем алгоритм будет постепенно улучшать значения в массиве dist, в результате получит кратчайшие расстояния до всех вершин.



В отличие от обычного алгоритма Дейкстры, становится ненужным поиск индекса. Изначально в queue помещаем стартовую вершину start с её расстоянием. Основной цикл алгоритма выполняется, пока в очереди есть хоть одна вершина. Из очереди извлекается вершина с наименьшим расстоянием, и затем из неё выполняются релаксации. Перед выполнением каждой успешной релаксации мы сначала удаляем из queue старую пару, а затем, после выполнения релаксации, добавляем обратно новую пару.



Главная задача функции info() состоит в выводе информации о графе, его диагонали, радиуса, периферийных и центральных вершинах. С помощью алгоритма Флойда-Уоршелла находим эксцентриситет вершины (расстояние до наиболее удаленной вершины графа). Зная эксцентриситет для каждой вершины, мы можем выдать информацию о графе.

**Радиусом графа** называется минимальный эксцентриситет среди всех вершин графа

**Диаметром графа** - это наибольшее расстояние между всеми парами вершин графа

**Центральной вершиной** графа является вершина чей эксцентриситет равен радиусу графа.

**Периферийной вершиной** графа является вершина чей эксцентриситет равен диаметру графа.

**Результат работы программы:**

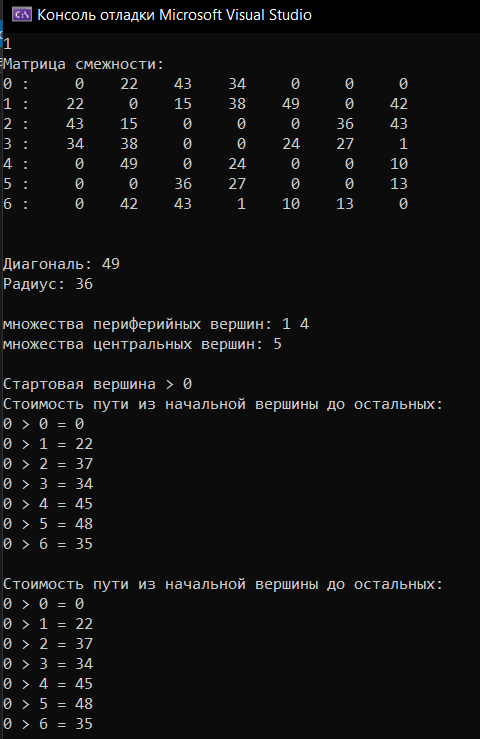


Рисунок 1- результат работы неориентированного взвешенного графа

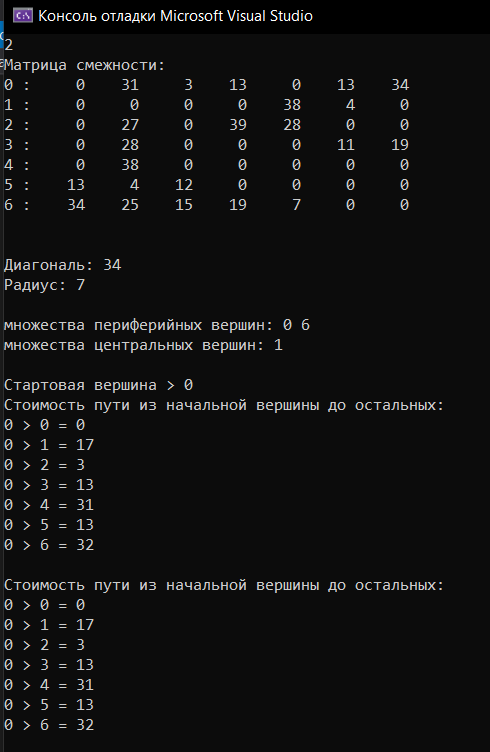


Рисунок 2- результат работы ориентированного взвешенного графа

**Листинг:**

**Dijkstra.cpp**

#include <iostream>

#include <iomanip>

#include <set>

#include <list>

#include <queue>

using namespace std;

class Graph {

private:

int\*\* G;

int numVertex;

public:

Graph(int numVertex)

{

this->numVertex = numVertex;

G = new int\* [numVertex];

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

G[i] = new int[numVertex];

for (int j = 0; j < numVertex; j++)

G[i][j] = 0;

}

}

~Graph() {

for (int i = 0; i < numVertex; i++)

delete[] G[i];

delete[] G;

}

void addEdge(int i, int j, int w, int u)

{

u==0 ? G[i][j]=w : u==1 ? G[j][i] = w : G[i][j] = G[j][i] = w;

}

void toString() {

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

cout << i << " : ";

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

cout << setw(5) << G[i][j] << " ";

}

cout << endl;

}

}

void Dijkstra(int st);

void shortestPath(int st);

void infograph();

};

void Graph::Dijkstra(int st)

{

int\* distance = new int[numVertex];

bool\* visited = new bool[numVertex];

int count, index;

for (int i = 0; i < numVertex; i++)

distance[i] = INT\_MAX, visited[i] = false;

distance[st] = 0;

for (count = 0; count < numVertex - 1; count++)

{

int min = INT\_MAX;

for (int i = 0; i < numVertex; i++)

if (!visited[i] && distance[i] <= min)

{

min = distance[i]; index = i;

}

int u = index;

visited[u] = true;

for (int i = 0; i < numVertex; i++)

if (!visited[i] && G[u][i] && distance[u] != INT\_MAX && distance[u] + G[u][i] < distance[i])

distance[i] = distance[u] + G[u][i];

}

cout << "Стоимость пути из начальной вершины до остальных:\t\n";

for (int i = 0; i < numVertex; i++)

if (distance[i] != INT\_MAX)

cout << st << " > " << i << " = " << distance[i] << endl;

else

cout << st << " > " << i << " = " << "маршрут недоступен" << endl;

delete[] distance;

delete[] visited;

}

void Graph::shortestPath(int st)

{

int\* dist = new int[numVertex];

int\* visit = new int[numVertex];

for (int i = 0; i < numVertex; i++)

visit[i] = false, dist[i] = INT\_MAX;

dist[st] = 0;

visit[st] = true;

queue<int>q;

q.push(st);

while (!q.empty()) {

int t = q.front();

q.pop();

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

if (!visit[i] && G[t][i] && G[t][i] + dist[t] < dist[i])

{ // Ставим условия заходим только если вершина не посещена, с этой вершиной есть ребро, расстояние между вершинами меньше чем уже существует

dist[i] = G[t][i] + dist[t];

q.push(i);

}

}

}

cout << "Стоимость пути из начальной вершины до остальных:\t\n";

for (int i = 0; i < numVertex; i++)

if (dist[i] != INT\_MAX)

cout << st << " > " << i << " = " << dist[i] << endl;

else

cout << st << " > " << i << " = " << "маршрут недоступен" << endl;

delete[] dist;

delete[] visit;

}

void Graph::infograph()

{

int\* ext= new int[numVertex];;

set <int> center, pereph;

int rad = INT\_MAX, diam;

int \*\*d= new int\* [numVertex];

for (int i = 0; i < numVertex; i++)

d[i] = new int[numVertex];

// Алгоритм Флойда-Уоршелла

for (int k = 0; k < numVertex; k++) {

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

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

d[i][j] = min(G[i][j], G[i][k] + G[k][j]);

}

}

}

// Нахождение эксцентриситета

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

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

ext[i] = max(ext[i], d[i][j]);

}

}

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

if (min(rad, ext[i])>0) {

rad = min(rad, ext[i]);

}

diam = max(diam, ext[i]);

}

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

if (ext[i] == rad) {

center.insert(i);

}

}

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

if (ext[i] == diam) {

pereph.insert(i);

}

}

cout << endl << "Диаметр: " << diam << endl << "Радиус: " << rad << endl;

cout << endl << "множества периферийных вершин: ";

for (int i : pereph)

cout << i<<" ";

cout << endl << "множества центральных вершин: ";

for (int i : center)

cout << i<<" ";

for (int i = 0; i < numVertex; i++)

delete[] d[i];

delete[] d;

delete[]ext;

}

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

{

setlocale(0,"");

srand(time(0));

int z = 7;

Graph c(z);

cout << argv[0]<<endl;

//int k, b; cin >> b >> k;

if (argv[argc-2]=="1" && argv[argc-1]=="1") {

//if (b == 1 && k == 1) {

for (int i = 0; i < z; i++)

{

for (int j = i + 1; j < z; j++)

{

if (0 < (rand() % 3)) {

c.addEdge(i, j, 1, 3);

}

}

}

}

if (argv[argc-2]=="1" && argv[argc-1]=="2") {

//if (b == 1 && k == 2) {

for (int i = 0; i < z; i++)

{

for (int j = i + 1; j < z; j++)

{

if (0 < (rand() % 3)) {

c.addEdge(i, j, 1, rand() % 3);

}

}

}

}

if (argv[argc-2]=="2" && argv[argc-1]=="1") {

//if (b==2 && k==1) {

for (int i = 0; i < z; i++)

{

for (int j = i + 1; j < z; j++)

{

if (0<(rand() % 3)) {

c.addEdge(i, j, rand() % 50, 3);

}

}

}

}

if (argv[argc-2]=="2" && argv[argc-1] == "2") {

//if (b==2 && k==2) {

for (int i = 0; i < z; i++)

{

for (int j = i + 1; j < z; j++)

{

if (0<(rand() % 3)) {

c.addEdge(i, j, rand() % 50, rand()%3);

}

}

}

}

cout << "Матрица смежности: \n";

c.toString();

cout << endl; c.infograph(); cout << endl;

int start;

cout << endl << "Стартовая вершина > "; cin >> start;

c.Dijkstra(start);

cout << endl;

c.shortestPath(start);

return 0;

}

**Вывод:** Мы научилисьрешать задачи, использующие графовые модели представления, изучили различные методов решения и анализы. Реализовалипоиск кратчайшего пути во взвешенном графе, алгоритм дейкстры.