Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

**ОТЧЕТ**

по лабораторной работе №10

по дисциплине «Логика и основы алгоритмизации в инженерных задачах»

# на тему: «Поиск расстояний во взвешенном графе»

Выполнили студенты группы 22ВВВ3:

Куракин Н.Н.

Майоров Н.А.

Матюшин К. М.

Приняли:

Юрова О.В.

Акифьев И.В.

Пенза 2023

**Название**

Поиск расстояний во взвешенном графе

**Цель работы -** Научиться искать расстояния во взвешенном графе.

**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).

#include <iostream>

#include <vector>

#include <queue>

#include <cstdlib>

using namespace std;

// Функция для создания случайного взвешенного графа

vector<vector<int>> generateRandomWeightedGraph(int n) {

vector<vector<int>> graph(n, vector<int>(n, 0));

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

for (int j = i + 1; j < n; ++j) {

if (rand() % 2 == 1) {

int weight = rand() % 10 + 1; // случайный вес от 1 до 10

graph[i][j] = weight;

graph[j][i] = weight;

}

}

}

return graph;

}

// Функция для вывода матрицы смежности графа

void printGraphMatrix(const vector<vector<int>>& graph) {

for (const auto& row : graph) {

for (int weight : row) {

cout << weight << ' ';

}

cout << '\n';

}

}

// Процедура поиска расстояний в неориентированном взвешенном графе

void bfsDistance(const vector<vector<int>>& graph, int start) {

int n = graph.size();

vector<int> distance(n, -1);

queue<int> q;

distance[start] = 0;

q.push(start);

while (!q.empty()) {

int current = q.front();

q.pop();

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

if (graph[current][neighbor] > 0 && (distance[neighbor] == -1 || distance[neighbor] > distance[current] + graph[current][neighbor])) {

distance[neighbor] = distance[current] + graph[current][neighbor];

q.push(neighbor);

}

}

}

// Вывод расстояний от start до остальных вершин

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

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

cout << "До вершины " << i << ": " << distance[i] << '\n';

}

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(0)); // Инициализация генератора случайных чисел

// Создание графа и вывод его матрицы смежности

int size;

cout << "Введите размер графа: ";

cin >> size;

int n = size; // Размер графа

if (size <= 0) {

cout << "Недопустимый размер графа";

return 0;

}

vector<vector<int>> randomWeightedGraph = generateRandomWeightedGraph(n);

cout << "Граф матрицы:\n";

printGraphMatrix(randomWeightedGraph);

// Выбор случайной вершины в качестве стартовой для поиска расстояний

int startVertex;

cout << "Стартовая вершина: ";

cin >> startVertex;

if (startVertex < 0 || startVertex >= size ) {

cout << "Нет такой вершины";

return 0;

}

cout << "\nВыбор стартовой вершины для поиска расстояния: " << startVertex << "\n\n";

// Поиск расстояний в графе и вывод результатов

bfsDistance(randomWeightedGraph, startVertex);

return 0;

}

Для ориентированного:

#include <iostream>

#include <vector>

#include <queue>

#include <cstdlib>

using namespace std;

// Функция для создания случайного ориентированного взвешенного графа

vector<vector<int>> generateRandomWeightedDirectedGraph(int n) {

vector<vector<int>> graph(n, vector<int>(n, 0));

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

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

if (i != j && rand() % 2 == 1) {

int weight = rand() % 10 + 1; // случайный вес от 1 до 10

graph[i][j] = weight;

}

}

}

return graph;

}

// Процедура поиска расстояний в ориентированном взвешенном графе

void bfsDistance(const vector<vector<int>>& graph, int start) {

int n = graph.size();

vector<int> distance(n, -1);

queue<int> q;

distance[start] = 0;

q.push(start);

while (!q.empty()) {

int current = q.front();

q.pop();

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

if (graph[current][neighbor] > 0 && (distance[neighbor] == -1 || distance[neighbor] > distance[current] + graph[current][neighbor])) {

distance[neighbor] = distance[current] + graph[current][neighbor];

q.push(neighbor);

}

}

}

// Вывод расстояний от start до остальных вершин

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

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

cout << "До вершины " << i << ": " << distance[i] << '\n';

}

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(0)); // Инициализация генератора случайных чисел

// Создание графа и вывод его матрицы смежности

int size;

cout << "Введите размер графа: ";

cin >> size;

int n = size; // Размер графа

if (size <= 0) {

cout << "Недопустимый размер графа";

return 0;

}

vector<vector<int>> randomWeightedDirectedGraph = generateRandomWeightedDirectedGraph(n);

cout << "Ориентированный граф:\n";

for (const auto& row : randomWeightedDirectedGraph) {

for (int weight : row) {

cout << weight << ' ';

}

cout << '\n';

}

// Выбор случайной вершины в качестве стартовой для поиска расстояний

int startVertex;

cout << "Стартовая вершина: ";

cin >> startVertex;

if (startVertex < 0 || startVertex >= size ) {

cout << "Нет такой вершины";

return 0;

}

cout << "\nВершина, от которой ищем расстояние: " << startVertex << "\n\n";

// Поиск расстояний в графе и вывод результатов

bfsDistance(randomWeightedDirectedGraph, startVertex);

return 0;

}

**Задание 2**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

const int INF = INT\_MAX;

// Функция для создания случайного ориентированного взвешенного графа

vector<vector<int>> generateRandomWeightedDirectedGraph(int n) {

vector<vector<int>> graph(n, vector<int>(n, 0));

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

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

if (i != j && rand() % 2 == 1) {

int weight = rand() % 10 + 1; // случайный вес от 1 до 10

graph[i][j] = weight;

}

}

}

return graph;

}

// Функция для нахождения радиуса, диаметра и центральных/периферийных вершин

void findRadiusDiameter(const vector<vector<int>>& graph, int& radius, int& diameter,

vector<int>& centralVertices, vector<int>& peripheralVertices) {

int n = graph.size();

radius = INF;

diameter = 0;

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

vector<int> distance(n, INF);

queue<int> q;

distance[startVertex] = 0;

q.push(startVertex);

while (!q.empty()) {

int current = q.front();

q.pop();

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

if (graph[current][neighbor] > 0 && distance[neighbor] == INF) {

distance[neighbor] = distance[current] + graph[current][neighbor];

q.push(neighbor);

}

}

}

int maxDistance = \*max\_element(distance.begin(), distance.end());

if (maxDistance < radius) {

radius = maxDistance;

centralVertices = { startVertex };

}

else if (maxDistance == radius) {

centralVertices.push\_back(startVertex);

}

diameter = max(diameter, maxDistance);

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

if (distance[i] == radius) {

peripheralVertices.push\_back(i);

}

}

}

radius = radius == INF ? 0 : radius;

diameter = diameter == INF ? 0 : diameter;

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(0)); // Инициализация генератора случайных чисел

int size;

cout << "Введите размер графа: ";

cin >> size;

int n = size; // Размер графа

if (size <= 0) {

cout << "Недопустимый размер графа";

return 0;

}

vector<vector<int>> randomWeightedDirectedGraph = generateRandomWeightedDirectedGraph(n);

cout << "Матрица ориентированного графа: \n";

for (const auto& row : randomWeightedDirectedGraph) {

for (int weight : row) {

cout << weight << ' ';

}

cout << '\n';

}

int radius, diameter;

vector<int> centralVertices, peripheralVertices;

// Находим радиус, диаметр, центральные и периферийные вершины

findRadiusDiameter(randomWeightedDirectedGraph, radius, diameter, centralVertices, peripheralVertices);

cout << "\nРадиус: " << radius << "\n";

cout << "Диаметр: " << diameter << "\n";

cout << "Центральная вершина: ";

for (int vertex : centralVertices) {

cout << vertex << " ";

}

cout << "\n";

cout << "Переферийные вершины: ";

for (int vertex : peripheralVertices) {

cout << vertex << " ";

}

cout << "\n";

return 0;

}

**Задание 3**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include <time.h>

#include <locale.h>

#define INF INT\_MAX

// Функция для создания случайного графа

int\*\* generateRandomGraph(int n, int weighted, int directed) {

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

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

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

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

graph[i][j] = 0;

if (i != j && (directed || i < j) && (rand() % 2 == 1 || !weighted)) {

int weight = (weighted) ? (rand() % 10 + 1) : (rand() % 2);

graph[i][j] = weight;

if (!directed) {

graph[j][i] = weight;

}

}

}

}

return graph;

}

// Функция для вывода матрицы смежности графа

void printGraphMatrix(int\*\* graph, int n) {

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

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

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

}

printf("\n");

}

}

// Процедура поиска расстояний в графе

void bfsDistance(int\*\* graph, int n, int start) {

int\* distance = (int\*)malloc(n \* sizeof(int));

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

distance[i] = INF;

}

int\* queue = (int\*)malloc(n \* sizeof(int));

int front = 0, rear = 0;

distance[start] = 0;

queue[rear++] = start;

while (front != rear) {

int current = queue[front++];

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

if (graph[current][neighbor] > 0 && (distance[neighbor] == INF || distance[neighbor] > distance[current] +

graph[current][neighbor])) {

distance[neighbor] = distance[current] + graph[current][neighbor];

queue[rear++] = neighbor;

}

}

}

// Вывод расстояний от start до остальных вершин

printf("Дистанция от вершины %d:\n", start);

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

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

}

free(distance);

free(queue);

}

int contains(int\* array, int size, int value) {

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

if (array[i] == value) {

return 1;

}

}

return 0;

}

// Функция для нахождения радиуса, диаметра и центральных/периферийных вершин

void findRadiusDiameter(int\*\* graph, int n, int\* radius, int\* diameter, int\* centralVertices, int\* centralVerticesCount, int\* peripheralVertices, int\* peripheralVerticesCount) {

\*radius = 0;

\*diameter = 0;

int\* distance = (int\*)malloc(n \* sizeof(int));

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

distance[i] = 0;

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

distance[i] = (graph[i][j] >= distance[i]) ? graph[i][j] : distance[i];

}

}

/\*distance[startVertex] = 0;

queue[rear++] = startVertex;

while (front != rear) {

int current = queue[front++];

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

if (graph[current][neighbor] > 0 && distance[neighbor] == INF) {

distance[neighbor] = distance[current] + graph[current][neighbor];

queue[rear++] = neighbor;

}

}

}\*/

// Используем переменную maxDiameter для отслеживания максимального диаметра

int maxDiameter = 0, minDiameter = INF;

// Находим максимальное расстояние внутри distance

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

if (distance[i] != INF && distance[i] != 0) {

maxDiameter = (maxDiameter > distance[i]) ? maxDiameter : distance[i];

minDiameter = (minDiameter <= distance[i]) ? minDiameter : distance[i];

}

}

// После завершения внутреннего цикла обновляем diameter

\*diameter = maxDiameter;

\*radius = \*radius == INF ? 0 : minDiameter;

// Находим центральные вершины

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

if (distance[i] == minDiameter && distance[i] != 0) {

centralVertices[(\*centralVerticesCount)++] = i;

}

if (distance[i] == maxDiameter && distance[i] != 0) {

peripheralVertices[(\*peripheralVerticesCount)++] = i;

}

}

}

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

setlocale(LC\_ALL, "RUS");

srand(time(0));

if (argc != 4) {

printf("Используйте: %s <graph\_type> <directed> <size>\n", argv[0]);

printf("Тип графа: weighted unweighted\n");

printf("Тип: 0 (undirected) or 1 (directed)\n");

return 1;

}

char\* graphType = argv[1];

int weighted = (strcmp(graphType, "weighted") == 0) ? 1 : 0;

int directed = atoi(argv[2]) == 1 ? 1 : 0;

int n = atoi(argv[3]);

int\*\* randomGraph = generateRandomGraph(n, weighted, 1);

printf("Матрица графа:\n");

printGraphMatrix(randomGraph, n);

int radius, diameter;

int\* centralVertices = (int\*)malloc(sizeof(int) \* n);

int\* peripheralVertices = (int\*)malloc(sizeof(int) \* n);

int centralVerticesCount = 0;

int peripheralVerticesCount = 0;

// Находим радиус, диаметр, центральные и пери ферийные вершины

findRadiusDiameter(randomGraph, n, &radius, &diameter, centralVertices, &centralVerticesCount, peripheralVertices,

&peripheralVerticesCount);

if (n <= 1 || radius == INF || radius == 0) {

printf("\nРадиус: Нет, ай ай ай\n");

printf("Диаметр: Нет, ой ой ой\n");

printf("Центральные вершины: \n");

printf("Периферийные вершины: \n");

return 0;

}

printf("\nРадиус: %d\n", radius == INF ? 0 : radius);

printf("Диаметр: %d\n", diameter == INF ? 0 : diameter);

printf("Центральные вершины: ");

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

printf("%d ", centralVertices[i]);

}

printf("\n");

printf("Периферийные вершины: ");

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

printf("%d ", peripheralVertices[i]);

}

printf("\n");

// Освобождаем выделенную память

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

free(randomGraph[i]);

}

free(randomGraph);

return 0;

}

**Вывод** – на примере лабораторной работы наша бригада научилась искать расстояния во взвешенном графе ориентированного и неориентированного вариантов.