Министерство науки и высшего образования РФ 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 24ВВВ4

Королёв Д.В Алешин К.А

Приняли: Юрова О.В Деев М.В

Пенза 2024

**Цель работы** – научиться на практике реализации алгоритма поиска расстояния в взвешенном графе

**Общие сведения** – Во взвешенном графе в отличие от не взвешенного каждое ребро имеет вес, отличный от нуля. Поэтому в матрице смежности взвешенного графа содержится информация не только о наличии ребра, но и о его весе. Поиск расстояний между вершинами в таком графе также возможно построить используя процедуры обхода графа. Отличие от поиска расстояний в не взвешенном графе будет состоять в том, что при обновлении расстояния до вершины при ее посещении оно будет увеличиваться не на 1, а на величину веса ребра. Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

**Листинг**

**Задание 1**

**1-2)**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <queue>

void generate\_adjacency\_matrix(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

int edge = rand() % 2;

if (edge == 1) {

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

matrix[j][i] = matrix[i][j];

}

}

}

}

void BFSD(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] > 0 && DIST[i] == -1) {

q.push(i);

DIST[i] = DIST[v] + matrix[v][i];

}

}

}

}

int main() {

setlocale(LC\_ALL, "Russian");

int n;

while (true) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(time(NULL));

generate\_adjacency\_matrix(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << "\t";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

**3)**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <queue>

void generate\_adjacency\_matrix(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

if (i != j) {

int edge = rand() % 2;

if (edge == 1) {

matrix[i][j] = rand() % 10 + 1; // Устанавливаем вес ребра

}

}

}

}

}

void BFSD(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] > 0 && DIST[i] == -1) {

q.push(i);

DIST[i] = DIST[v] + matrix[v][i];

}

}

}

}

int main() {

setlocale(LC\_ALL, "Russian");

int n;

while (true) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(time(NULL));

generate\_adjacency\_matrix(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << "\t";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

**Задание 2**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <limits.h>

#include <locale.h>

void generate\_adjacency\_matrix\_oriented(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

if (i != j) {

int edge = rand() % 2;

if (edge == 1) {

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

}

}

}

}

}

void generate\_adjacency\_matrix\_undirected(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

int edge = rand() % 2;

if (edge == 1) {

int weight = rand() % 10 + 1;

matrix[i][j] = weight;

matrix[j][i] = weight;

}

}

}

}

void dijkstra(int\*\* matrix, int n, int start, int\* distances) {

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

distances[i] = INT\_MAX;

}

distances[start] = 0;

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

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

visited[i] = 0;

}

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

int min\_distance = INT\_MAX;

int min\_index;

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

if (visited[v] == 0 && distances[v] <= min\_distance) {

min\_distance = distances[v];

min\_index = v;

}

}

visited[min\_index] = 1;

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

if (!visited[i] && matrix[min\_index][i] > 0 &&

distances[min\_index] != INT\_MAX &&

distances[min\_index] + matrix[min\_index][i] < distances[i]) {

distances[i] = distances[min\_index] + matrix[min\_index][i];

}

}

}

free(visited);

}

void calculate\_shortest\_paths(int\*\* matrix, int n, int\*\* shortest\_paths) {

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

dijkstra(matrix, n, i, shortest\_paths[i]);

}

}

void calculate\_radius\_diameter(int\*\* matrix, int n, int\* radius, int\* diameter) {

\*radius = INT\_MAX;

\*diameter = 0;

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

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

dijkstra(matrix, n, i, distances);

int max\_distance = 0;

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

if (distances[j] < INT\_MAX) {

max\_distance = (max\_distance > distances[j]) ? max\_distance : distances[j];

}

}

if (max\_distance < \*radius) {

\*radius = max\_distance;

}

if (max\_distance > \*diameter) {

\*diameter = max\_distance;

}

free(distances);

}

}

void calculate\_central\_and\_peripheral\_vertices(int\*\* matrix, int n) {

int\* max\_distances = (int\*)malloc(n \* sizeof(int));

int min\_max\_distance = INT\_MAX;

int max\_distance = 0;

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

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

dijkstra(matrix, n, i, distances);

max\_distance = 0;

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

if (distances[j] < INT\_MAX) {

max\_distance = (max\_distance > distances[j]) ? max\_distance : distances[j];

}

}

max\_distances[i] = max\_distance;

if (max\_distance < min\_max\_distance) {

min\_max\_distance = max\_distance;

}

free(distances);

}

printf("Центральные вершины: ");

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

if (max\_distances[i] == min\_max\_distance) {

printf("%d ", i);

}

}

printf("\n");

printf("Периферийные вершины: ");

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

if (max\_distances[i] == max\_distance) {

printf("%d ", i);

}

}

printf("\n");

free(max\_distances);

}

int main() {

setlocale(LC\_ALL, "Rus");

srand((unsigned)time(0));

int n;

printf("Введите количество вершин графа: ");

scanf("%d", &n);

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

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

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

}

char choice;

printf("Сгенерировать ориентированный (o) или неориентированный (u) граф? ");

scanf(" %c", &choice);

if (choice == 'o') {

generate\_adjacency\_matrix\_oriented(n, matrix);

}

else if (choice == 'u') {

generate\_adjacency\_matrix\_undirected(n, matrix);

}

else {

fprintf(stderr, "Ошибка: неверный ввод.\n");

return 1;

}

printf("Матрица смежности:\n");

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

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

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

}

printf("\n");

}

//матрица кратчайших расстояний

int\*\* shortest\_paths = (int\*\*)malloc(n \* sizeof(int\*));

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

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

}

calculate\_shortest\_paths(matrix, n, shortest\_paths);

printf("Матрица кратчайших расстояний:\n");

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

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

if (shortest\_paths[i][j] == INT\_MAX) {

printf("INF\t");

}

else {

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

}

}

printf("\n");

}

int radius, diameter;

calculate\_radius\_diameter(matrix, n, &radius, &diameter);

printf("Радиус графа: %d\n", radius);

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

calculate\_central\_and\_peripheral\_vertices(matrix, n);

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

free(matrix[i]);

free(shortest\_paths[i]);

}

free(matrix);

free(shortest\_paths);

return 0;

}

**Задание 3**

#include <stdio.h>

#include <string.h>

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <queue>

void generate\_adjacency\_matrix\_weighted\_directed(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

if (i != j) {

int edge = rand() % 2;

if (edge == 1) {

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

}

}

}

}

}

void generate\_adjacency\_matrix\_weighted\_undirected(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

int edge = rand() % 2;

if (edge == 1) {

int weight = rand() % 10 + 1; // weight between 1 and 10

matrix[i][j] = weight;

matrix[j][i] = weight; // undirected

}

}

}

}

void generate\_adjacency\_matrix\_unweighted\_undirected(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

int edge = rand() % 2;

matrix[i][j] = edge;

matrix[j][i] = edge;

}

}

}

void generate\_adjacency\_matrix\_unweighted\_directed(int n, int\*\* matrix) {

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

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

matrix[i][j] = 0;

}

}

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

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

if (i != j) {

int edge = rand() % 2;

if (edge == 1) {

matrix[i][j] = 1;

}

}

}

}

}

void BFSD\_weighted\_undirected(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] > 0 && DIST[i] == -1) {

q.push(i);

DIST[i] = DIST[v] + matrix[v][i];

}

}

}

}

void BFSD\_weighted\_directed(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] > 0 && DIST[i] == -1) {

q.push(i);

DIST[i] = DIST[v] + matrix[v][i];

}

}

}

}

void BFSD\_unweighted\_undirected(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] == 1 && DIST[i] == -1) {

DIST[i] = DIST[v] + 1;

q.push(i);

}

}

}

}

void BFSD\_unweighted\_directed(int start, int n, int\*\* matrix, int\* DIST) {

std::queue<int> q;

q.push(start);

DIST[start] = 0;

while (!q.empty()) {

int v = q.front();

q.pop();

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

if (matrix[v][i] == 1 && DIST[i] == -1) {

q.push(i);

DIST[i] = DIST[v] + 1;

}

}

}

}

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

setlocale(LC\_ALL, "Russian");

if (argc == 3) {

if (strcmp(argv[1], "weighted") == 0) { // граф взвешенный

if (strcmp(argv[2], "undirected") == 0) { // и неориентированный

int n;

while (1) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(time(NULL));

generate\_adjacency\_matrix\_weighted\_undirected(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << "\t";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD\_weighted\_undirected(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

else if (strcmp(argv[2], "directed") == 0) { // и ориентированный

int n;

while (1) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(time(NULL));

generate\_adjacency\_matrix\_weighted\_directed(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << "\t";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD\_weighted\_directed(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

}

else if (strcmp(argv[1], "unweighted") == 0) { // граф не взвешенный

if (strcmp(argv[2], "undirected") == 0) { // и не ориентированный

int n;

while (1) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(time(NULL));

generate\_adjacency\_matrix\_unweighted\_undirected(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << " ";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD\_unweighted\_undirected(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

else if (strcmp(argv[2], "directed") == 0) { // и ориентированный

int n;

while (true) {

std::cout << "Введите размер матрицы: ";

std::cin >> n;

if (n <= 0) {

break;

}

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

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

matrix[i] = new int[n];

}

int\* DIST = new int[n];

srand(static\_cast<unsigned int>(time(NULL)));

generate\_adjacency\_matrix\_unweighted\_directed(n, matrix);

std::cout << "Матрица смежности:\n";

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

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

std::cout << matrix[i][j] << "\t";

}

std::cout << "\n";

}

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

DIST[i] = -1;

}

int start;

std::cout << "Введите вершину, с которой начнется обход: ";

std::cin >> start;

BFSD\_unweighted\_directed(start, n, matrix, DIST);

std::cout << "Расстояния от вершины " << start << ": \n";

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

if (DIST[i] == -1) {

std::cout << "Вершина " << i << ": недоступна\n";

}

else {

std::cout << "Вершина " << i << ": " << DIST[i] << "\n";

}

}

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

delete[] matrix[i];

}

delete[] matrix;

delete[] DIST;

}

return 0;

}

}

}

else {

printf("аргументы не были переданы!");

return 0;

}

}

Рисунок 1 – Результат выполнения 1-го задания 1-2 часть

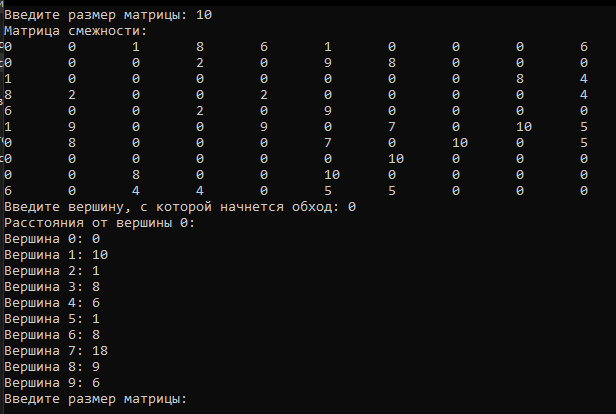


Рисунок 2 – Результат выполнения 1-го задания 3 часть

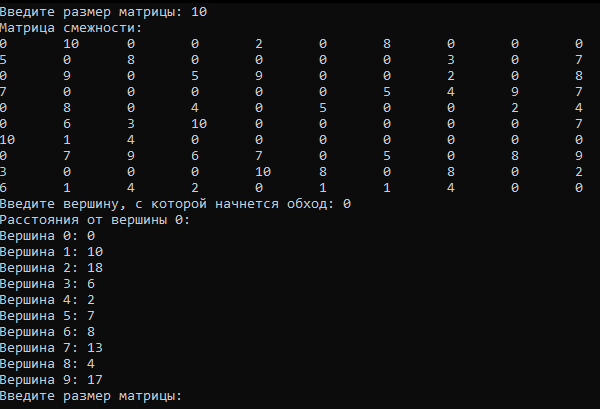


Рисунок 3 – Результат выполнения 2-го задания

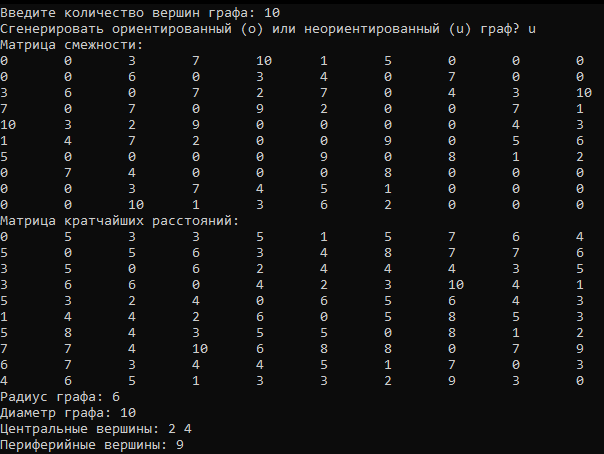
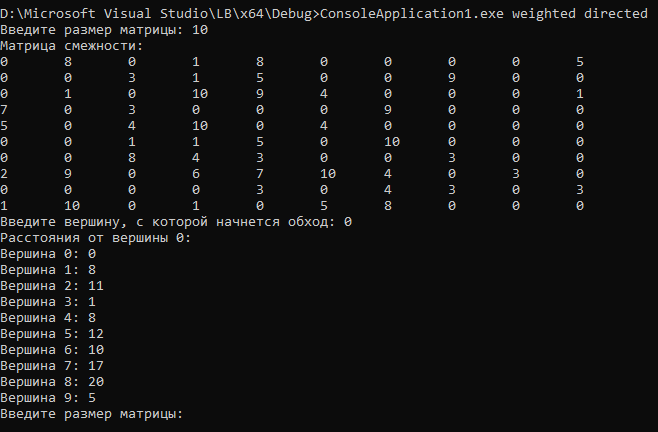


Рисунок 4 – Результат выполнения 3-го задания



**Вывод**: В ходе выполнения лабораторной работы были изучены на практике навыки реализации алгоритма поиска путей в взвешенном графе.