Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

*по лабораторной работе №7*

*По дисциплине: «Логика и основы алгоритмизации в инженерных задачах»*

*На тему: «Поиск расстояний во взвешенном графе»*

**Выполнил студенты группы 19ВВ3:**

Ерёмин А.А

**Принял:**

Митрохин М.А

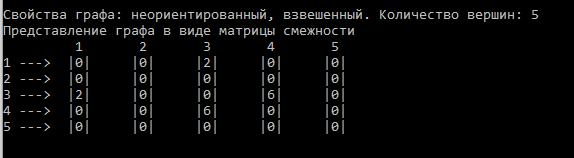
Пенза 2020

**Цель:** изучить и реализовать алгоритмы поиска расстояний во взвешенном графе.

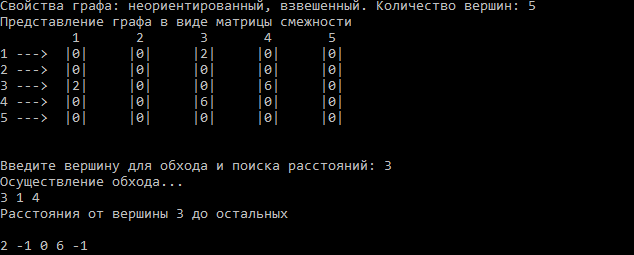
**Ход работы:**

**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++.



**Листинг (1.2):**

void BFSDWeight(Graph\* graph, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

dist[start\_vertex] = 0;

Q.push(start\_vertex);

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d ", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] > dist[current\_vertex] + graph->matrix[current\_vertex][i]) {

Q.push(i);

dist[i] = dist[current\_vertex] + graph->matrix[current\_vertex][i];

}

}

}

}

**Генерация взвешенной матрицы:**

void GraphUndirectWeight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j)

GraphG->matrix[i][j] = 0;

else if (GetAdj())

GraphG->matrix[i][j] = GraphG->matrix[j][i] = rand() % 10;

else

GraphG->matrix[i][j] = GraphG->matrix[j][i] = 0;

}

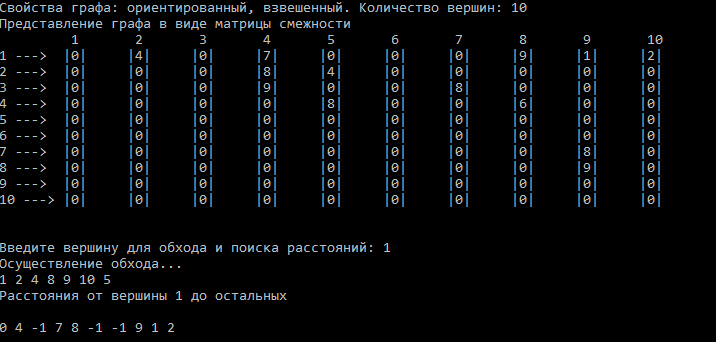
}

}

GetAdj() – функция случайного появления смежности.

Взвешенный граф – это граф, в котором каждому ребру сопоставляется какое-то число – длина пути по этому ребру. Из этого следует вывод о том, что простое прибавление 1, как это было в невзвешенном графе не позволит определить расстояние от определенной вершины до всех остальных. Для того, чтобы реализовать такой алгоритм именно во взвешенном графе, стоит изменить условие пометки для не посещенной вершины, т.к граф взвешенный, то матрица смежности будет отражать не только наличие пути, но и расстояние ребра и достаточно просто изменить условие dist[i] = dist[v] + 1 на dist[i] = dist[v] + G[v][i]. Также используется условие dist[i] > dist[v] + G[v][i] для поиска кратчайших путей, чтобы оно работало, вместо обозначения не посещенной вершины как -1, нужно присвоить вектору расстояний максимальные значения(Например, MAX\_INT).

**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.



**Листинг (1.3):**

void BFSDWeight(Graph\* graph, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

dist[start\_vertex] = 0;

Q.push(start\_vertex);

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d ", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] > dist[current\_vertex] + graph->matrix[current\_vertex][i]) {

Q.push(i);

dist[i] = dist[current\_vertex] + graph->matrix[current\_vertex][i];

}

}

}

}

**Генерация матрицы:**

void GraphDirectWeight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j)

GraphG->matrix[i][j] = 0;

else if (GetAdj()) {

GraphG->matrix[i][j] = rand() % 10;

GraphG->matrix[j][i] = 0;

}

else

GraphG->matrix[i][j] = GraphG->matrix[j][i] = 0;

}

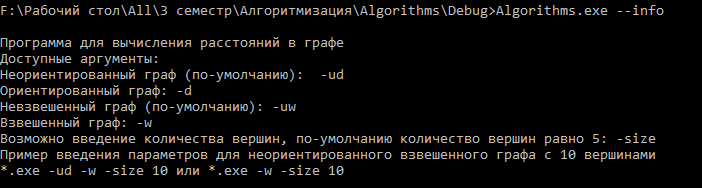
}

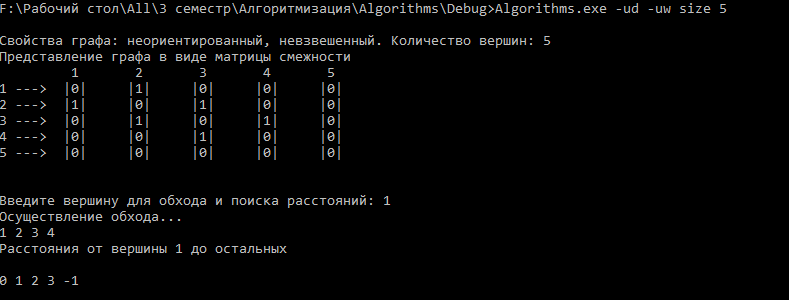
}

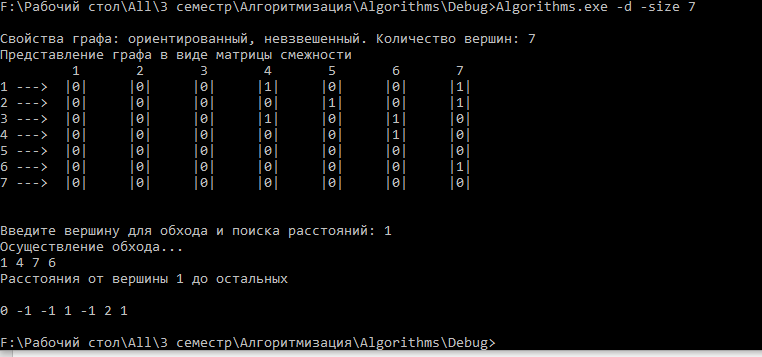
Можно попасть из одной вершины в другую, но не обратно.

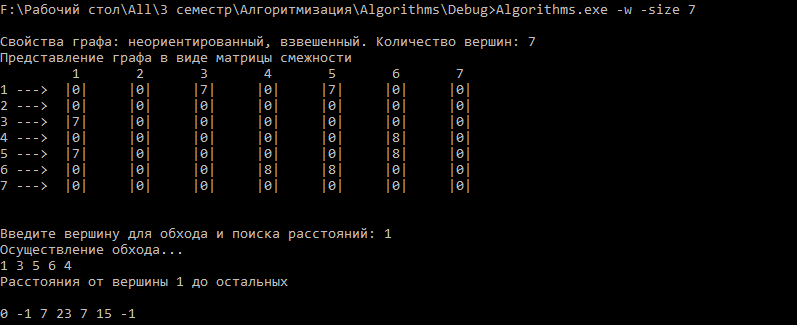
**Задание 2\***

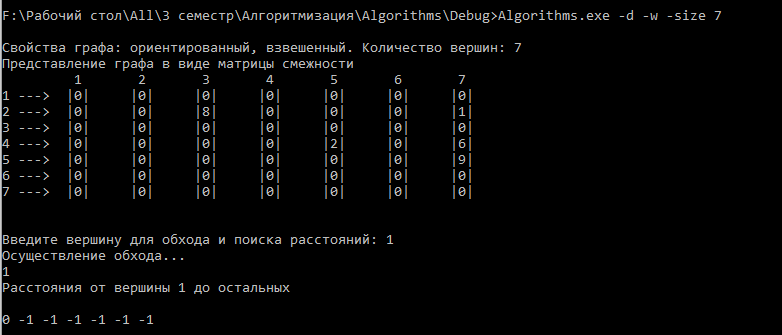
**1**. Модернизируйте программу так, чтобы получить возможность запуска программы с параметрами командной строки (см. описание ниже). В качестве параметра должны указываться тип графа (взвешенный или нет) и наличие ориентации его ребер (есть ориентация или нет).











**Вывод:** в ходе выполнения лабораторных заданий познакомился с алгоритмом поиска расстояний во взвешенном графе и реализовал его на языке си.

**Листинг (all):**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Algorithms.h"

typedef struct Graph {

int\*\* matrix;

int\* vertexes;

int matrix\_order;

};

typedef struct Node {

int vertex;

Node\* next;

};

typedef struct Lists {

int num\_vertexes;

Node\*\* head;

};

int\*\* MatrixMalloc(int matrix\_order) {

int\*\* Matrix = (int\*\*)malloc(sizeof(int) \* matrix\_order);

for (int i = 0; i < matrix\_order; i++) {

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

}

return Matrix;

}

int\* VertexesMalloc(int matrix\_order) {

int\* Vertexes = (int\*)malloc(sizeof(int) \* matrix\_order);

for (int i = 0; i < matrix\_order; i++) {

Vertexes[i] = i + 1;

}

return Vertexes;

}

Graph\* GraphCreate(int matrix\_order) {

Graph\* pGraph = (Graph\*)malloc(sizeof(Graph));

pGraph->matrix = MatrixMalloc(matrix\_order);

pGraph->vertexes = VertexesMalloc(matrix\_order);

pGraph->matrix\_order = matrix\_order;

return pGraph;

}

int\* ArrayCreate(int num\_vertexes) {

int\* Array = (int\*)malloc(sizeof(int) \* num\_vertexes);

return Array;

}

void DistanceArrayRefresh(int\* distance\_array, int num\_vertexes) {

for (int i = 0; i < num\_vertexes; i++) {

distance\_array[i] = -1;

}

}

void DistanceArrayMax(int\* distance\_array, int num\_vertexes) {

for (int i = 0; i < num\_vertexes; i++) {

distance\_array[i] = INT\_MAX;

}

}

void VisitedArrayRefresh(int\* visited\_array, int num\_vertexes) {

for (int i = 0; i < num\_vertexes; i++) {

visited\_array[i] = 0;

}

}

void GraphFree(Graph\* GraphG) {

int MatrixOrder = GraphG->matrix\_order;

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

free(GraphG->matrix[i]);

}

free(GraphG->matrix);

free(GraphG->vertexes);

free(GraphG);

}

int GetAdj() {

if (rand() % 101 <= 30) {

return 1;

}

else {

return 0;

}

}

void GraphUndirectUnweight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j) {

GraphG->matrix[i][j] = 0;

}

else {

GraphG->matrix[i][j] = GraphG->matrix[j][i] = GetAdj();

}

}

}

}

void GraphUndirectWeight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j)

GraphG->matrix[i][j] = 0;

else if (GetAdj())

GraphG->matrix[i][j] = GraphG->matrix[j][i] = rand() % 10;

else

GraphG->matrix[i][j] = GraphG->matrix[j][i] = 0;

}

}

}

void GraphDirectWeight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j)

GraphG->matrix[i][j] = 0;

else if (GetAdj()) {

GraphG->matrix[i][j] = rand() % 10;

GraphG->matrix[j][i] = 0;

}

else

GraphG->matrix[i][j] = GraphG->matrix[j][i] = 0;

}

}

}

void GraphDirectUnweight(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j)

GraphG->matrix[i][j] = 0;

else if (GetAdj()) {

GraphG->matrix[i][j] = 1;

GraphG->matrix[j][i] = 0;

}

else

GraphG->matrix[i][j] = GraphG->matrix[j][i] = 0;

}

}

}

void MatrixPrint(Graph\* GraphG) {

printf("\n");

int matrix\_order = GraphG->matrix\_order;

printf("\t");

for (int i = 0; i < matrix\_order; i++) {

printf(" %d\t", GraphG->vertexes[i]);

}

printf("\n");

for (int i = 0; i < matrix\_order; i++) {

printf("%d --->\t", GraphG->vertexes[i]);

for (int j = 0; j < matrix\_order; j++) {

printf("|%d|\t", GraphG->matrix[i][j]);

}

printf("\n");

}

printf("\n");

}

void ArrayPrint(int\* array, int size) {

printf("\n");

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

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

}

}

void DistanceArrayPrint(int\* array, int size) {

printf("\n");

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

if (array[i] == INT\_MAX) {

printf("-1 ");

}

else {

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

}

}

}

Node\* NodeCreate(int vertex) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

Lists\* ListsCreate(int num\_vertexes) {

Lists\* list = (Lists\*)malloc(sizeof(struct Lists));

list->num\_vertexes = num\_vertexes;

list->head = (Node \* \*)malloc(sizeof(struct Node\*) \* num\_vertexes);

for (int i = 0; i < num\_vertexes; i++) {

list->head[i] = NULL;

}

return list;

}

void ListsDelete(Lists\* list) {

for (int i = 0; i < list->num\_vertexes; i++) {

while (list->head[i] != NULL) {

Node\* prevNode = list->head[i];

list->head[i] = list->head[i]->next;

free(prevNode);

}

}

free(list->head);

free(list);

}

void AddFirst(Lists\* list, Node\* newNode, int vertex) {

newNode->next = list->head[vertex];

list->head[vertex] = newNode;

}

void AddAfter(Node\* prevNode, Node\* newNode) {

newNode->next = prevNode->next;

prevNode->next = newNode;

}

void AddLast(Lists\* list, Node\* newNode, int vertex) {

Node\* current = list->head[vertex];

if (current == NULL) {

AddFirst(list, newNode, vertex);

return;

}

while (current->next) {

current = current->next;

}

AddAfter(current, newNode);

}

void TransMatrixInList(Graph\* graph, Lists\* list) {

int matrix\_order = graph->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = 0; j < matrix\_order; j++) {

if (graph->matrix[i][j] == 1) {

Node\* newNode = NodeCreate(j);

AddLast(list, newNode, i);

}

}

}

}

void ListsPrint(Lists\* list) {

printf("\n");

printf("\nСписок смежности");

printf("\n");

Node\* current;

for (int i = 0; i < list->num\_vertexes; i++) {

printf("%d", i + 1);

current = list->head[i];

while (current != NULL) {

printf("->%d", current->vertex + 1);

current = current->next;

}

printf("\n");

}

}

void BFSDWeight(Graph\* graph, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

dist[start\_vertex] = 0;

Q.push(start\_vertex);

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d ", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] > dist[current\_vertex] + graph->matrix[current\_vertex][i]) {

Q.push(i);

dist[i] = dist[current\_vertex] + graph->matrix[current\_vertex][i];

}

}

}

}

void BFSD(Graph\* graph, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

dist[start\_vertex] = 0;

Q.push(start\_vertex);

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d ", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] == -1) {

Q.push(i);

dist[i] = dist[current\_vertex] + graph->matrix[current\_vertex][i];

}

}

}

}

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

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

const char PROGRAMinfo[] = "--info";

const char UNDIRECTgraph[] = "-ud";

const char DIRECTgraph[] = "-d"; // Это лучше как-то объединить

const char UNWEIGHTgraph[] = "-uw";

const char WEIGHTgraph[] = "-w";

const char SIZEgraph[] = "-size";

const int MAXarg = 5;

if (argc == 2) {

if (strcmp(argv[1], PROGRAMinfo) == 0) {

printf("\n");

printf("Программа для вычисления расстояний в графе\n");

printf("Доступные аргументы: \n");

printf("Неориентированный граф (по-умолчанию): ");

puts(UNDIRECTgraph);

printf("Ориентированный граф: ");

puts(DIRECTgraph);

printf("Невзвешенный граф (по-умолчанию): ");

puts(UNWEIGHTgraph);

printf("Взвешенный граф: ");

puts(WEIGHTgraph);

printf("Возможно введение количества вершин, по-умолчанию количество вершин равно 5: ");

puts(SIZEgraph);

printf("Пример введения параметров для неориентированного взвешенного графа с 10 вершинами\n");

printf("\*.exe -ud -w -size 10 или \*.exe -w -size 10");

printf("\n");

return 0;

}

}

if (argc >= 1 && argc <= MAXarg) {

int graph\_order = 5;

char current\_weight[] = "-uw";

char current\_direct[] = "-ud";

for (int i = 1; i < argc; i++) {

bool incurrect = true;

if (strcmp(argv[i], SIZEgraph) == 0 || strcmp(argv[i], UNDIRECTgraph) == 0 || strcmp(argv[i], DIRECTgraph) == 0 || strcmp(argv[i], UNWEIGHTgraph) == 0 || strcmp(argv[i], WEIGHTgraph) == 0 || atoi(argv[i]) != 0) {

incurrect = false;

}

if (incurrect) {

printf("Введенные аргументы неверны! Для просмотра доступных аргументов введите \*.exe --info");

return 1;

}

}

for (int i = 1; i < argc; i++) {

if (strcmp(argv[i], SIZEgraph) == 0) {

if (atoi(argv[i + 1]) == 0) {

printf("Введенные аргументы неверны! Для просмотра доступных аргументов введите \*.exe --info");

return 1;

}

graph\_order = atoi(argv[i + 1]);

}

if (strcmp(argv[i], DIRECTgraph) == 0) {

strcpy(current\_direct, "-d");

}

if (strcmp(argv[i], WEIGHTgraph) == 0) {

strcpy(current\_weight, "-w");

}

}

if (strcmp(current\_direct, UNDIRECTgraph) == 0) {

if (strcmp(current\_weight, UNWEIGHTgraph) == 0)

{

printf("\n");

printf("Свойства графа: неориентированный, невзвешенный. Количество вершин: %d\n", graph\_order);

Graph\* graph = GraphCreate(graph\_order);

GraphUndirectUnweight(graph);

printf("Представление графа в виде матрицы смежности");

MatrixPrint(graph);

int\* distance\_array = ArrayCreate(graph\_order);

int start\_vertex;

DistanceArrayRefresh(distance\_array, graph\_order);

printf("\n");

printf("Введите вершину для обхода и поиска расстояний: ");

scanf("%d", &start\_vertex);

start\_vertex--;

if (start\_vertex < 0 || start\_vertex > graph\_order - 1) {

printf("Неверный ввод стартовой вершины!");

return 1;

}

printf("Осуществление обхода...");

printf("\n");

BFSD(graph, distance\_array, start\_vertex);

printf("\n");

printf("Расстояния от вершины %d до остальных\n", start\_vertex + 1);

ArrayPrint(distance\_array, graph\_order);

printf("\n");

return 0;

}

}

if (strcmp(current\_direct, DIRECTgraph) == 0) {

if (strcmp(current\_weight, UNWEIGHTgraph) == 0) {

printf("\n");

printf("Свойства графа: ориентированный, невзвешенный. Количество вершин: %d\n", graph\_order);

Graph\* graph = GraphCreate(graph\_order);

GraphDirectUnweight(graph);

printf("Представление графа в виде матрицы смежности");

MatrixPrint(graph);

int\* distance\_array = ArrayCreate(graph\_order);

int start\_vertex;

DistanceArrayRefresh(distance\_array, graph\_order);

printf("\n");

printf("Введите вершину для обхода и поиска расстояний: ");

scanf("%d", &start\_vertex);

start\_vertex--;

if (start\_vertex < 0 || start\_vertex > graph\_order - 1) {

printf("Неверный ввод стартовой вершины!");

return 1;

}

printf("Осуществление обхода...");

printf("\n");

BFSD(graph, distance\_array, start\_vertex);

printf("\n");

printf("Расстояния от вершины %d до остальных\n", start\_vertex + 1);

ArrayPrint(distance\_array, graph\_order);

printf("\n");

}

}

if (strcmp(current\_direct, UNDIRECTgraph) == 0) {

if (strcmp(current\_weight, WEIGHTgraph) == 0)

{

printf("\n");

printf("Свойства графа: неориентированный, взвешенный. Количество вершин: %d\n", graph\_order);

Graph\* graph = GraphCreate(graph\_order);

GraphUndirectWeight(graph);

printf("Представление графа в виде матрицы смежности");

MatrixPrint(graph);

int\* distance\_array = ArrayCreate(graph\_order);

int start\_vertex;

DistanceArrayMax(distance\_array, graph\_order);

printf("\n");

printf("Введите вершину для обхода и поиска расстояний: ");

scanf("%d", &start\_vertex);

start\_vertex--;

if (start\_vertex < 0 || start\_vertex > graph\_order - 1) {

printf("Неверный ввод стартовой вершины!");

return 1;

}

printf("Осуществление обхода...");

printf("\n");

BFSDWeight(graph, distance\_array, start\_vertex);

printf("\n");

printf("Расстояния от вершины %d до остальных\n", start\_vertex + 1);

DistanceArrayPrint(distance\_array, graph\_order);

printf("\n");

return 0;

}

}

if (strcmp(current\_direct, DIRECTgraph) == 0) {

if (strcmp(current\_weight, WEIGHTgraph) == 0)

{

printf("\n");

printf("Свойства графа: ориентированный, взвешенный. Количество вершин: %d\n", graph\_order);

Graph\* graph = GraphCreate(graph\_order);

GraphDirectWeight(graph);

printf("Представление графа в виде матрицы смежности");

MatrixPrint(graph);

int\* distance\_array = ArrayCreate(graph\_order);

int start\_vertex;

DistanceArrayMax(distance\_array, graph\_order);

printf("\n");

printf("Введите вершину для обхода и поиска расстояний: ");

scanf("%d", &start\_vertex);

start\_vertex--;

if (start\_vertex < 0 || start\_vertex > graph\_order - 1) {

printf("Неверный ввод стартовой вершины!");

return 1;

}

printf("Осуществление обхода...");

printf("\n");

BFSDWeight(graph, distance\_array, start\_vertex);

printf("\n");

printf("Расстояния от вершины %d до остальных\n", start\_vertex + 1);

DistanceArrayPrint(distance\_array, graph\_order);

printf("\n");

return 0;

}

}

}

else {

printf("Введено неправильное количество аргументов! Для просмотра доступных аргументов введите \*.exe --info");

return 1;

}

}