МИНИСТЕРВСТВО 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 24ВВВ3:

Пяткин Р. С.

Гусаров Е. Е.

Принял:

к.т.н., доцент, Юрова О. В.

к.т.н., Деев М. В.

Пенза 2025

Цель

Изучение поиска расстояний 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 C++.

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.

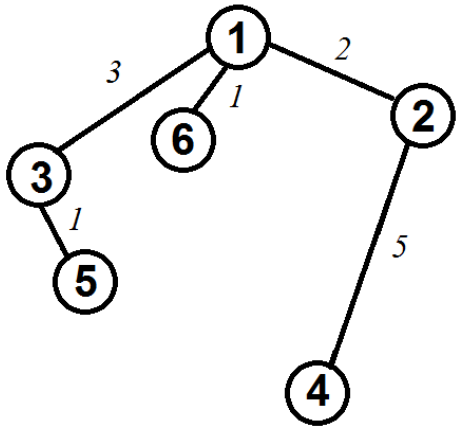
2. Определите подмножества периферийных и центральных вершин.

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).

Теория

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.



$$G = \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 1 \\ 2 & 0 & 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Рисунок 1 – Граф

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.

Реализация

Вход: G – матрица смежности графа, v – исходная вершина.

Выход: DIST – вектор расстояний до всех вершин от исходной.

Алгоритм ПОШ

1.1. для всех i положим DIST [i] = -1 пометим как "не посещенную";

1.2. ВЫПОЛНЯТЬ BFSD (v).

1.3 для всех i вывести DIST [i] на экран;

Алгоритм BFSD(v):

2.1. Создать пустую очередь Q = {};

2.2. Поместить v в очередь Q.push(v);

2.3. Обновить вектор расстояний DIST [x] = 0;

2.4. ПОКА Q != Æ очередь не пуста ВЫПОЛНЯТЬ

2.5. v = Q.front() установить текущую вершину;

2.6. Удалить первый элемент из очереди Q.pop();

2.7. вывести на экран v;

2.8. ДЛЯ i = 1 ДО size_G ВЫПОЛНЯТЬ

2.9. ЕСЛИ G(v,i) > 0 И DIST = = -1

2.10. ТО

2.11. Поместить i в очередь Q.push(i);

2.12. Обновить вектор расстояний DIST [i] = DIST [v] + G(v,i);

Реализация состоит из подготовительной части, в которой все вершины помечаются как не посещенные (п.1.1). Не посещенные вершины помечаются – 1, т.к. значение 0 и 1 могут быть расстояниями. Расстояние 0 – от исходной вершины до самой себя.

В самой процедуре сначала создается пустая очередь (п. 2.1), в которую помещается исходная вершина, из которой начат обход (п.2.2). Расстояние

до этой вершины (п.2.3) устанавливается равным 0 (расстояние до самой себя).

Далее итерационно, пока очередь не опустеет, из нее извлекается первый элемент, который становится текущей вершиной (п. 2.5, 2.6). Затем в цикле просматривается v-я строка матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с v не посещенную вершину (п.2.9), эта вершина помещается в очередь (п.2.11) и для нее обновляется вектор расстояния (п.2.12). Расстояние до новой i-й вершины вычисляется как расстояние до текущей v-й вершины плюс вес ребра до новой вершины G(v,i).

После просмотра строки матрицы смежности алгоритм делает следующую итерацию цикла 2.4 или заканчивает работу, если очередь пуста.

Если для всех пар вершин графа определены расстояния, то можно вычислить эксцентриситет

Если G - граф, содержащий непустое множество n вершин V и множество ребер E и d(vi, vj) – расстояние между двумя произвольными вершинами vi и vj, тогда для фиксированной вершины v величина

,

где v, vj V и j = 1…n называется эксцентриситетом вершины vi.

Другими словами эксцентриситет вершины – расстояние до наиболее удаленной вершины графа.

Максимальный эксцентриситет среди эксцентриситетов всех вершин графа называется диаметром графа G и обозначается через D(G).

Вершина vi называется периферийной, если её эксцентриситет равен диаметру графа e(vi) = d(G).

Минимальный из эксцентриситетов вершин графа называется его радиусом и обозначается через r(G).

Вершина vi называется центральной, если её эксцентриситет равен радиусу графа e(vi) = r(G).

Множество всех центральных вершин графа называется его центром. Граф G может иметь единственную центральную вершину или несколько центральных вершин.

Вывод:

В ходе выполнения лабораторной работы были разработаны программа для выполнения заданий Лабораторной работы №10. В процессе выполнения работы был изучен поиск расстояний во взвешенном графе.

Листинг

Файл lab_10.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>
#include <limits.h>

using namespace std;

int printMatrix(int** m, int n) {
cout << " ";
for (int i = 0; i < n; i++) {
cout << i << " ";
}
cout << endl;

for (int i = 0; i < n; i++) {
cout << i << "| ";
for (int j = 0; j < n; j++) {
if (m[i][j] == INT_MAX) {
cout << "∞ ";
} else {
cout << m[i][j] << " ";
}
}
cout << endl;
}
cout << endl;
return 0;
}

int** createMatrix(int n) {
int** m = new int*[n];
for (int i = 0; i < n; i++) {
m[i] = new int[n];
}

for (int i = 0; i < n; i++) {
for (int j = i; j < n; j++) {
if (i == j) {
m[i][j] = 0;
```

```
} else {
bool hasEdge = (rand() % 100) < 70;
if (hasEdge) {
m[i][j] = m[j][i] = rand() % 11;
} else {
m[i][j] = m[j][i] = 0;
}
}
}
}
return m;
}

void deleteMatrix(int** m, int n) {
for (int i = 0; i < n; ++i) {
delete[] m[i];
}
delete[] m;
}

void BFSD(int** G, int numG, int** GD, int start) {
int* dist = new int[numG];
bool* visited = new bool[numG];

for (int i = 0; i < numG; i++) {
dist[i] = INT_MAX;
visited[i] = false;
}

dist[start] = 0;

priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
pq.push(make_pair(0, start));

while (!pq.empty()) {
int u = pq.top().second;
pq.pop();

if (visited[u]) continue;
visited[u] = true;

for (int v = 0; v < numG; v++) {
if (G[u][v] > 0) {
if (!visited[v] && dist[u] != INT_MAX) {
int newDist = dist[u] + G[u][v];
if (newDist < dist[v]) {
dist[v] = newDist;
pq.push(make_pair(dist[v], v));
}
}
}
}
```

```
}
}

// Записываем результаты в матрицу расстояний
for (int i = 0; i < numG; i++) {
GD[start][i] = (dist[i] == INT_MAX ? -1 : dist[i]);
}

delete[] dist;
delete[] visited;
}

// Функция для вычисления эксцентриситетов
int* calculateEccentricities(int** GD, int n) {
int* ecc = new int[n];
for (int i = 0; i < n; i++) {
ecc[i] = 0;
}

for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (GD[i][j] > ecc[i] && GD[i][j] != -1) {
ecc[i] = GD[i][j];
}
}
}

return ecc;
}

int main() {
srand(time(0));

int numG = 0;

cout << "Введите количество вершин в графе: ";
cin >> numG;

if (numG <= 0) {
cout << "Количество вершин должно быть положительным числом!" << endl;
return 1;
}

cout << "\n=== Задача 1: Генерация матрицы смежности ===" << endl;
int** G = createMatrix(numG);
cout << "Матрица смежности графа (0 означает отсутствие ребра):" << endl;
printMatrix(G, numG);

cout << "\n=== Задача 2: Поиск расстояний с использованием BFS/очереди ===" << endl;
int** GD = new int*[numG];
for (int i = 0; i < numG; i++) {
```

```
GD[i] = new int[numG];
for (int j = 0; j < numG; j++) {
GD[i][j] = -1;
}
}

for (int i = 0; i < numG; i++) {
BFSD(G, numG, GD, i);
}

cout << "Матрица кратчайших расстояний (-1 означает недостижимость):" << endl;
printMatrix(GD, numG);

cout << "\n=== Дополнительная информация ===" << endl;
int* ecc = calculateEccentricities(GD, numG);
cout << "Эксцентриситеты вершин:" << endl;
for (int i = 0; i < numG; i++) {
cout << "Вершина " << i << ": " << ecc[i] << endl;
}

int radius = INT_MAX;
int diameter = 0;
for (int i = 0; i < numG; i++) {
if (ecc[i] < radius) radius = ecc[i];
if (ecc[i] > diameter) diameter = ecc[i];
}

cout << "\nРадиус графа: " << radius << endl;
cout << "Диаметр графа: " << diameter << endl;

cout << "Центральные вершины: ";
for (int i = 0; i < numG; i++) {
if (ecc[i] == radius) {
cout << i << " ";
}
}
cout << endl;

cout << "Периферийные вершины: ";
for (int i = 0; i < numG; i++) {
if (ecc[i] == diameter) {
cout << i << " ";
}
}
cout << endl;

deleteMatrix(G, numG);
deleteMatrix(GD, numG);
delete[] ecc;

return 0;
```

```
}
```

Файл lab_10_2.cpp

```
#include <iostream>
#include <ctime>
#include <vector>
#include <queue>
#include <limits>
#include <iomanip>
#include <algorithm>

using namespace std;

const int INF = numeric_limits<int>::max();

vector<vector<int>> createWeightedDirectedGraph(int n, int maxWeight = 15) {
vector<vector<int>> graph(n, vector<int>(n, 0));
for (int i = 0; i < n; i++) {
int j = (i + 1) % n;
int weight = 1 + rand() % maxWeight;
graph[i][j] = weight;
}
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (i != j && rand() % 3 == 1) {
int weight = 1 + rand() % maxWeight;
graph[i][j] = weight;
}
}
}
return graph;
}

void printAdjacencyMatrix(const vector<vector<int>>& graph) {
int n = graph.size();
cout << "Матрица смежности ориентированного графа:" << endl;
cout << " ";
for (int i = 0; i < n; i++) {
cout << setw(3) << i;
}
cout << endl;
for (int i = 0; i < n; i++) {
cout << setw(2) << i << ":";
for (int j = 0; j < n; j++) {
if (graph[i][j] == 0 && i != j) {
cout << setw(3) << "-";
} else {
cout << setw(3) << graph[i][j];
}
```

```
}
cout << endl;
}
cout << endl;
}

vector<vector<int>> floydWarshall(const vector<vector<int>>& graph) {
int n = graph.size();
vector<vector<int>> dist(n, vector<int>(n, INF));
for (int i = 0; i < n; i++) {
dist[i][i] = 0;
for (int j = 0; j < n; j++) {
if (graph[i][j] > 0) {
dist[i][j] = graph[i][j];
}
}
}
for (int k = 0; k < n; k++) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (dist[i][k] != INF && dist[k][j] != INF) {
if (dist[i][j] > dist[i][k] + dist[k][j]) {
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}
}
}
return dist;
}

void printShortestDistances(const vector<vector<int>>& dist) {
int n = dist.size();
cout << "== Задача 2: Поиск расстояний с использованием BFS/очереди ==" << endl;
cout << "Матрица кратчайших расстояний (-1 означает недостижимость):" << endl << endl;
cout << "| |";
for (int j = 0; j < n; j++) {
cout << setw(3) << j << " |";
}
cout << endl;
cout << "|";
for (int j = 0; j <= n; j++) {
cout << "----|";
}
cout << endl;
for (int i = 0; i < n; i++) {
cout << "| " << setw(2) << i << " |";
for (int j = 0; j < n; j++) {
if (dist[i][j] == INF) {
cout << setw(3) << -1 << " |";
} else {
```

```
cout << setw(3) << dist[i][j] << " |";
}
}
cout << endl;
}
cout << endl;
}

void calculateGraphProperties(const vector<vector<int>>& dist) {
int n = dist.size();
vector<int> eccentricity(n, 0);
for (int i = 0; i < n; i++) {
int maxDist = 0;
bool hasUnreachable = false;
for (int j = 0; j < n; j++) {
if (i != j) {
if (dist[i][j] == INF) {
hasUnreachable = true;
} else if (dist[i][j] > maxDist) {
maxDist = dist[i][j];
}
}
}
if (hasUnreachable) {
eccentricity[i] = INF;
} else {
eccentricity[i] = maxDist;
}
}
int radius = INF;
int diameter = 0;
for (int i = 0; i < n; i++) {
if (eccentricity[i] != INF) {
if (eccentricity[i] < radius) {
radius = eccentricity[i];
}
if (eccentricity[i] > diameter) {
diameter = eccentricity[i];
}
}
}
if (radius == INF) {
radius = -1;
}
if (diameter == 0) {
diameter = -1;
}
vector<int> centralVertices;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == radius) {
centralVertices.push_back(i);
```

```
}
}
vector<int> peripheralVertices;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == diameter) {
peripheralVertices.push_back(i);
}
}
cout << "== Дополнительная информация ==" << endl;
cout << "Эксцентриситеты вершин:" << endl;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == INF) {
cout << "Вершина " << i << ": недостижимы некоторые вершины (∞)" << endl;
} else {
cout << "Вершина " << i << ": " << eccentricity[i] << endl;
}
}
cout << endl;
if (radius == -1 || diameter == -1) {
cout << "Граф не является сильно связным. Радиус и диаметр не определены." << endl;
} else {
cout << "Радиус графа: " << radius << endl;
cout << "Диаметр графа: " << diameter << endl;
}
if (!centralVertices.empty()) {
cout << "Центральные вершины: ";
for (size_t i = 0; i < centralVertices.size(); i++) {
cout << centralVertices[i];
if (i < centralVertices.size() - 1) {
cout << " ";
}
}
cout << endl;
}
if (!peripheralVertices.empty()) {
cout << "Периферийные вершины: ";
for (size_t i = 0; i < peripheralVertices.size(); i++) {
cout << peripheralVertices[i];
if (i < peripheralVertices.size() - 1) {
cout << " ";
}
}
cout << endl << endl;
}
}

int main() {
srand(time(0));
int n;
cout << "АНАЛИЗ ОРИЕНТИРОВАННОГО ВЗВЕШЕННОГО ГРАФА" << endl;
cout << "==========================================" << endl;
```

```
cout << "Введите количество вершин в графе: ";
cin >> n;
if (n <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}
cout << "\nСоздание ориентированного взвешенного графа из " << n << " вершин..." << endl;
auto graph = createWeightedDirectedGraph(n, 15);
cout << "\n";
printAdjacencyMatrix(graph);
auto distMatrix = floydWarshall(graph);
printShortestDistances(distMatrix);
calculateGraphProperties(distMatrix);
cout << "\n========================================" << endl;
cout << "Анализ графа завершен!" << endl;
return 0;
}
```

Файл lab_10_3.cpp

```
#include <iostream>
#include <ctime>
#include <vector>
#include <queue>
#include <limits>
#include <iomanip>
#include <algorithm>
#include <cstring>
#include <cstdlib>

using namespace std;

const int INF = numeric_limits<int>::max();

struct ProgramOptions {
int vertices = 0;
bool isWeighted = true;
bool isDirected = true;
bool showHelp = false;
int maxWeight = 15;
};

ProgramOptions parseArguments(int argc, char* argv[]) {
ProgramOptions options;
for (int i = 1; i < argc; i++) {
if (strcmp(argv[i], "-n") == 0 && i + 1 < argc) {
options.vertices = atoi(argv[i + 1]);
i++;
} else if (strcmp(argv[i], "-t") == 0 && i + 1 < argc) {
if (strcmp(argv[i + 1], "weighted") == 0) {
```

```
options.isWeighted = true;
} else if (strcmp(argv[i + 1], "unweighted") == 0) {
options.isWeighted = false;
} else {
cerr << "Ошибка: неизвестный тип графа: " << argv[i + 1] << endl;
cerr << "Допустимые значения: weighted, unweighted" << endl;
exit(1);
}
i++;
} else if (strcmp(argv[i], "-o") == 0 && i + 1 < argc) {
if (strcmp(argv[i + 1], "directed") == 0) {
options.isDirected = true;
} else if (strcmp(argv[i + 1], "undirected") == 0) {
options.isDirected = false;
} else {
cerr << "Ошибка: неизвестная ориентация: " << argv[i + 1] << endl;
cerr << "Допустимые значения: directed, undirected" << endl;
exit(1);
}
i++;
} else if (strcmp(argv[i], "-w") == 0 && i + 1 < argc) {
options.maxWeight = atoi(argv[i + 1]);
if (options.maxWeight <= 0) {
cerr << "Ошибка: максимальный вес должен быть положительным" << endl;
exit(1);
}
i++;
} else if (strcmp(argv[i], "-h") == 0) {
options.showHelp = true;
} else {
cerr << "Ошибка: неизвестный параметр: " << argv[i] << endl;
exit(1);
}
}
return options;
}

vector<vector<int>> createGraph(const ProgramOptions& options) {
int n = options.vertices;
vector<vector<int>> graph(n, vector<int>(n, 0));
for (int i = 0; i < n; i++) {
int j = (i + 1) % n;
int weight = options.isWeighted ? (1 + rand() % options.maxWeight) : 1;
graph[i][j] = weight;
if (!options.isDirected) {
graph[j][i] = weight;
}
}
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (i != j && rand() % 3 == 1) {
```

```
int weight = options.isWeighted ? (1 + rand() % options.maxWeight) : 1;
graph[i][j] = weight;
if (!options.isDirected) {
graph[j][i] = weight;
}
}
}
}
return graph;
}

void printAdjacencyMatrix(const vector<vector<int>>& graph, const ProgramOptions& options) {
int n = graph.size();
cout << "Матрица смежности ";
if (options.isDirected) {
cout << "ориентированного ";
} else {
cout << "неориентированного ";
}
if (options.isWeighted) {
cout << "взвешенного ";
} else {
cout << "невзвешенного ";
}
cout << "графа:" << endl;
cout << " ";
for (int i = 0; i < n; i++) {
cout << setw(3) << i;
}
cout << endl;
for (int i = 0; i < n; i++) {
cout << setw(2) << i << ":";
for (int j = 0; j < n; j++) {
if (graph[i][j] == 0 && i != j) {
cout << setw(3) << "-";
} else {
cout << setw(3) << graph[i][j];
}
}
cout << endl;
}
cout << endl;
}

vector<vector<int>> floydWarshall(const vector<vector<int>>& graph) {
int n = graph.size();
vector<vector<int>> dist(n, vector<int>(n, INF));
for (int i = 0; i < n; i++) {
dist[i][i] = 0;
for (int j = 0; j < n; j++) {
if (graph[i][j] > 0) {
```

```
dist[i][j] = graph[i][j];
}
}
}
for (int k = 0; k < n; k++) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
if (dist[i][k] != INF && dist[k][j] != INF) {
if (dist[i][j] > dist[i][k] + dist[k][j]) {
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}
}
}
}
return dist;
}

void printShortestDistances(const vector<vector<int>>& dist) {
int n = dist.size();
cout << "== Задача 2: Поиск расстояний с использованием BFS/очереди ==" << endl;
cout << "Матрица кратчайших расстояний (-1 означает недостижимость):" << endl << endl;
cout << "| |";
for (int j = 0; j < n; j++) {
cout << setw(3) << j << " |";
}
cout << endl;
cout << "|";
for (int j = 0; j <= n; j++) {
cout << "----|";
}
cout << endl;
for (int i = 0; i < n; i++) {
cout << "| " << setw(2) << i << " |";
for (int j = 0; j < n; j++) {
if (dist[i][j] == INF) {
cout << setw(3) << -1 << " |";
} else {
cout << setw(3) << dist[i][j] << " |";
}
}
cout << endl;
}
cout << endl;
}

void calculateGraphProperties(const vector<vector<int>>& dist) {
int n = dist.size();
vector<int> eccentricity(n, 0);
for (int i = 0; i < n; i++) {
int maxDist = 0;
```

```
bool hasUnreachable = false;
for (int j = 0; j < n; j++) {
if (i != j) {
if (dist[i][j] == INF) {
hasUnreachable = true;
} else if (dist[i][j] > maxDist) {
maxDist = dist[i][j];
}
}
}
if (hasUnreachable) {
eccentricity[i] = INF;
} else {
eccentricity[i] = maxDist;
}
}
int radius = INF;
int diameter = 0;
for (int i = 0; i < n; i++) {
if (eccentricity[i] != INF) {
if (eccentricity[i] < radius) {
radius = eccentricity[i];
}
if (eccentricity[i] > diameter) {
diameter = eccentricity[i];
}
}
}
if (radius == INF) {
radius = -1;
}
if (diameter == 0) {
diameter = -1;
}
vector<int> centralVertices;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == radius) {
centralVertices.push_back(i);
}
}
vector<int> peripheralVertices;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == diameter) {
peripheralVertices.push_back(i);
}
}
cout << "== Дополнительная информация ==" << endl;
cout << "Эксцентриситеты вершин:" << endl;
for (int i = 0; i < n; i++) {
if (eccentricity[i] == INF) {
cout << "Вершина " << i << ": недостижимы некоторые вершины (∞)" << endl;
```

```
} else {
cout << "Вершина " << i << ": " << eccentricity[i] << endl;
}
}
cout << endl;
if (radius == -1 || diameter == -1) {
cout << "Граф не является сильно связным. Радиус и диаметр не определены." << endl;
} else {
cout << "Радиус графа: " << radius << endl;
cout << "Диаметр графа: " << diameter << endl;
}
if (!centralVertices.empty()) {
cout << "Центральные вершины: ";
for (size_t i = 0; i < centralVertices.size(); i++) {
cout << centralVertices[i];
if (i < centralVertices.size() - 1) {
cout << " ";
}
}
cout << endl;
}
if (!peripheralVertices.empty()) {
cout << "Периферийные вершины: ";
for (size_t i = 0; i < peripheralVertices.size(); i++) {
cout << peripheralVertices[i];
if (i < peripheralVertices.size() - 1) {
cout << " ";
}
}
cout << endl << endl;
}
}

ProgramOptions interactiveInput() {
ProgramOptions options;
cout << "=== ИНТЕРАКТИВНЫЙ РЕЖИМ ===" << endl;
cout << "Введите количество вершин в графе: ";
cin >> options.vertices;
if (options.vertices <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
exit(1);
}
char choice;
cout << "Граф взвешенный? (y/n, по умолчанию y): ";
cin >> choice;
if (choice == 'n' || choice == 'N') {
options.isWeighted = false;
}
if (options.isWeighted) {
cout << "Введите максимальный вес ребра (по умолчанию 15): ";
string weightInput;
```

```
cin >> weightInput;
if (!weightInput.empty()) {
options.maxWeight = stoi(weightInput);
}
}
cout << "Граф ориентированный? (y/n, по умолчанию y): ";
cin >> choice;
if (choice == 'n' || choice == 'N') {
options.isDirected = false;
}
return options;
}

int main(int argc, char* argv[]) {
srand(time(0));
ProgramOptions options;
if (argc > 1) {
options = parseArguments(argc, argv);
if (options.showHelp) {
return 0;
}
if (options.vertices == 0) {
cout << "Ошибка: не указано количество вершин!" << endl;
cout << "Используйте параметр -n <число>" << endl;
cout << endl;
return 1;
}
} else {
options = interactiveInput();
}
cout << endl;
cout << "=== ПАРАМЕТРЫ ПРОГРАММЫ ===" << endl;
cout << "Количество вершин: " << options.vertices << endl;
cout << "Тип графа: " << (options.isWeighted ? "взвешенный" : "невзвешенный") << endl;
if (options.isWeighted) {
cout << "Максимальный вес ребра: " << options.maxWeight << endl;
}
cout << "Ориентация: " << (options.isDirected ? "ориентированный" : "неориентированный") << endl;
cout << endl;
cout << "Создание графа из " << options.vertices << " вершин..." << endl;
auto graph = createGraph(options);
printAdjacencyMatrix(graph, options);
auto distMatrix = floydWarshall(graph);
printShortestDistances(distMatrix);
calculateGraphProperties(distMatrix);
return 0;
}
```