МИНИСТЕРВСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №9

по дисциплине: "Логика и основы алгоритмизации в инженерных задачах"

на тему: "Поиск расстояний в 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 в графе.

Лабораторное задание

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного 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. * Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

Задание 2 *

Реализуйте процедуру поиска расстояний на основе обхода в глубину.

Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.

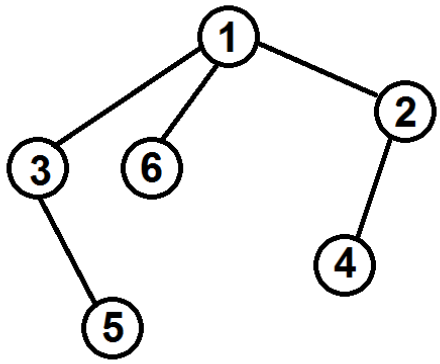
Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

Теория

Поиск расстояний – довольно распространенная задача анализа графов.

Для поиска расстояний можно использовать процедуры обхода графа. Для этого при каждом переходе в новую вершину необходимо запоминать,

сколько шагов до нее мы сделали. При этом вектор, который хранил информацию о посещении вершин становится вектором расстояний. Довольно просто модернизировать для поиска расстояний в графе алгоритм обхода в ширину, т.к. этот алгоритм проходит вершины по уровням удаленности, то для не ориентированного графа для вершин каждого следующего уровня глубины расстояние от исходной вершины увеличивается на 1. Удалённость в данном случае понимается как количество ребер, по которым необходимо прейти до достижения вершины.



$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

Рисунок 1 – Граф

Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

Реализация

Вход: G – матрица смежности графа.

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) = = 1И DIST = = -1

2.10. ТО

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

2.12. Обновить вектор расстояний DIST [i] = DIST [v] + 1;

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

В самой процедуре как и в алгоритме BFS 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 1 (так как ребра нашего графа не взвешенные).

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

Таким образом, если вершина помещается в очередь при просмотре сроки матрицы смежности на 1-й итерации, то они находятся на 1 уровне удаленности и расстояние до этих вершин будет равным 1.

DIST [i] = DIST [v] + 1, где DIST [v] = 0 – расстояние от исходной вершины до самой себя.

Далее, начинают просматриваться вершины первого уровня и соответствующие им строки матрицы смежности. При добавлении смежных с вершинами первого уровня вершин, расстояния до них будут равны 2.

DIST [i] = DIST [v] + 1, где DIST [v] = 1 – расстояние от исходной вершины до вершин 1 уровня.

После того, как все вершины первого уровня будут просмотрены и извлечены из очереди, начнется просмотр вершин 2 уровня и соответствующих им строк матрицы смежности. При добавлении смежных с вершинами второго уровня вершин, расстояния до них будут равны 3.

DIST [i] = DIST [v] + 1, где DIST [v] = 2 – расстояние от исходной вершины до вершин 2 уровня.

И так далее, алгоритм проходит вершины по уровням, пока очередь не опустеет.

Вывод:

В ходе выполнения лабораторной работы были разработаны программа для выполнения заданий Лабораторной работы №8. В процессе выполнения работы был изучен способ обхода графа в ширину.

Листинг

Файл lab_9.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>

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++){
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++){
m[i][j] = m[j][i] = (i == j ? 0 : rand() % 2);
}
}
return m;
}
void deleteM(int** m, int n, int* v) {
for (int i = 0; i < n; ++i) {
delete[] m[i];
}
delete[] m;
delete[] v;
}

void BFSD(int** G, int numG, int* distance, int s) {
queue<int> q;
int v = 0;
distance[s] = 0;
q.push(s);

while (!q.empty())
{
v = q.front();
q.pop();
cout<<v;

for(int i = 0; i < numG; i++){
if(G[v][i] == 1 && distance[i] == -1){
q.push(i);
distance[i] = distance[v] + 1;
}
}
}
cout<<endl;
cout<<"Distance from "<<s<<" to: "<<endl;

for(int i = 0; i < numG; i++){
cout<<i<<" : "<<distance[i]<<endl;
}
}
```

```
int main() {
srand(time(0));
int numG = 0;
int n = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}
int* distance = new int[numG];
for(int i = 0; i < numG; i++){
distance[i] = -1;
}

int** G = createMatrix(numG);

cout << "Матрица смежности графа:" << endl;
printMatrix(G, numG);

cout << "Введите начальную вершину обхода: ";
cin >> n;
BFSD(G, numG, distance, n);
cout << endl;
deleteM(G, numG, distance);
return 0;
}
```

Файл lab_9_2.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>

using namespace std;

struct ListNode {
int vertex;
ListNode* next;
ListNode(int v) : vertex(v), next(nullptr) {}
};

struct AdjacencyList {
ListNode* head;
int size;
AdjacencyList() : head(nullptr), size(0) {}
void add(int v) {
ListNode* newNode = new ListNode(v);
newNode->next = head;
```

```
head = newNode;
size++;
}
bool contains(int v) {
ListNode* current = head;
while (current != nullptr) {
if (current->vertex == v) {
return true;
}
current = current->next;
}
return false;
}
~AdjacencyList() {
ListNode* current = head;
while (current != nullptr) {
ListNode* next = current->next;
delete current;
current = next;
}
head = nullptr;
size = 0;
}
};

void printAdjacencyLists(AdjacencyList* graph, int n) {
cout << "Списки смежности графа:" << endl;
for (int i = 0; i < n; i++) {
cout << "Вершина " << i << ": ";
ListNode* current = graph[i].head;
while (current != nullptr) {
cout << current->vertex << " ";
current = current->next;
}
cout << endl;
}
cout << endl;
}

AdjacencyList* createGraph(int n) {
AdjacencyList* graph = new AdjacencyList[n];
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (rand() % 2 == 1) {
graph[i].add(j);
graph[j].add(i);
}
}
}
return graph;
}
```

```
void deleteGraph(AdjacencyList* graph, int n, int* distance) {
delete[] graph;
delete[] distance;
}

void BFSD(AdjacencyList* graph, int numG, int* distance, int s) {
queue<int> q;
int v = 0;
distance[s] = 0;
q.push(s);

while (!q.empty()) {
v = q.front();
q.pop();
cout << v << " ";

ListNode* current = graph[v].head;
while (current != nullptr) {
int neighbor = current->vertex;
if (distance[neighbor] == -1) {
q.push(neighbor);
distance[neighbor] = distance[v] + 1;
}
current = current->next;
}
}
cout << endl << "Расстояния от вершины " << s << " до:" << endl;
for (int i = 0; i < numG; i++) {
cout << "Вершина " << i << ": " << distance[i] << endl;
}
}

int main() {
srand(time(0));
int numG = 0;
int startVertex = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}
int* distance = new int[numG];
for (int i = 0; i < numG; i++) {
distance[i] = -1;
}

AdjacencyList* graph = createGraph(numG);

printAdjacencyLists(graph, numG);
```

```
cout << "Введите начальную вершину обхода: ";
cin >> startVertex;
if (startVertex < 0 || startVertex >= numG) {
cout << "Ошибка: неверный номер вершины!" << endl;
deleteGraph(graph, numG, distance);
return 1;
}
BFSD(graph, numG, distance, startVertex);
cout << endl;
deleteGraph(graph, numG, distance);
return 0;
}
```

Файл lab_9_3_1.cpp

```
#include <iostream>
#include <ctime>
#include <random>

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++){
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++){
m[i][j] = m[j][i] = (i == j ? 0 : rand() % 2);
}
}
return m;
```

```
}

void deleteM(int** m, int n, int* v) {
for (int i = 0; i < n; ++i) {
delete[] m[i];
}
delete[] m;
if (v != nullptr) {
delete[] v;
}
}

void DFSD(int** G, int numG, int* visited, int* distance, int current, int currentDistance) {
visited[current] = 1;
distance[current] = currentDistance;
cout << current << " ";
for(int i = 0; i < numG; i++){
if(G[current][i] == 1 && visited[i] == 0){
DFSD(G, numG, visited, distance, i, currentDistance + 1);
}
}
}

void DFSDistance(int** G, int numG, int startVertex) {
int* visited = new int[numG];
int* distance = new int[numG];
for(int i = 0; i < numG; i++){
visited[i] = 0;
distance[i] = -1;
}
DFSD(G, numG, visited, distance, startVertex, 0);
cout << endl;
cout << "Расстояния от вершины " << startVertex << " до:" << endl;
for(int i = 0; i < numG; i++){
if(distance[i] != -1) {
cout << "Вершина " << i << ": " << distance[i] << endl;
} else {
cout << "Вершина " << i << ": недостижима" << endl;
}
}
delete[] visited;
delete[] distance;
}

int main(){
srand(time(0));
int numG, current = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
```

```
return 1;
}

int** G = createMatrix(numG);
printMatrix(G, numG);

cout << "Введите начальную вершину обхода: ";
cin >> current;
if (current < 0 || current >= numG) {
cout << "Ошибка: неверный номер вершины!" << endl;
deleteM(G, numG, nullptr);
return 1;
}
DFSDistance(G, numG, current);
deleteM(G, numG, nullptr);
return 0;
}
```

Файл lab_9_3_2.cpp

```
#include <iostream>
#include <ctime>
#include <random>

using namespace std;

struct ListNode {
int vertex;
ListNode* next;
ListNode(int v) : vertex(v), next(nullptr) {}
};

struct AdjacencyList {
ListNode* head;
int size;
AdjacencyList() : head(nullptr), size(0) {}
void add(int v) {
ListNode* newNode = new ListNode(v);
newNode->next = head;
head = newNode;
size++;
}
bool contains(int v) {
ListNode* current = head;
while (current != nullptr) {
if (current->vertex == v) {
return true;
}
current = current->next;
}
return false;
```

```
}
~AdjacencyList() {
ListNode* current = head;
while (current != nullptr) {
ListNode* next = current->next;
delete current;
current = next;
}
head = nullptr;
size = 0;
}
};

void printAdjacencyLists(AdjacencyList* graph, int n) {
cout << "Списки смежности графа:" << endl;
for (int i = 0; i < n; i++) {
cout << "Вершина " << i << ": ";
ListNode* current = graph[i].head;
while (current != nullptr) {
cout << current->vertex << " ";
current = current->next;
}
cout << endl;
}
cout << endl;
}

AdjacencyList* createGraph(int n) {
AdjacencyList* graph = new AdjacencyList[n];
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (rand() % 2 == 1) {
graph[i].add(j);
graph[j].add(i);
}
}
}
return graph;
}

// Удаление графа
void deleteGraph(AdjacencyList* graph, int n, int* distance, int* visited) {
delete[] graph;
delete[] distance;
if (visited != nullptr) {
delete[] visited;
}
}

void DFSDRecursive(AdjacencyList* graph, int numG, int* visited, int* distance,
int current, int currentDistance) {
```

```
visited[current] = 1;
distance[current] = currentDistance;
cout << current << " ";
ListNode* neighbor = graph[current].head;
while (neighbor != nullptr) {
int neighborVertex = neighbor->vertex;
if (visited[neighborVertex] == 0) {
DFSDRecursive(graph, numG, visited, distance,
neighborVertex, currentDistance + 1);
}
neighbor = neighbor->next;
}
}

void DFSDistance(AdjacencyList* graph, int numG, int startVertex) {
int* visited = new int[numG];
int* distance = new int[numG];
for (int i = 0; i < numG; i++) {
visited[i] = 0;
distance[i] = -1;
}
DFSDRecursive(graph, numG, visited, distance, startVertex, 0);
cout << endl;
cout << "Расстояния от вершины " << startVertex << " до:" << endl;
for (int i = 0; i < numG; i++) {
if (distance[i] != -1) {
cout << "Вершина " << i << ": " << distance[i] << endl;
} else {
cout << "Вершина " << i << ": недостижима" << endl;
}
}
delete[] visited;
delete[] distance;
}

int main() {
srand(time(0));
int numG = 0;
int startVertex = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}

AdjacencyList* graph = createGraph(numG);
printAdjacencyLists(graph, numG);

cout << "Введите начальную вершину обхода: ";
cin >> startVertex;
```

```
if (startVertex < 0 || startVertex >= numG) {
cout << "Ошибка: неверный номер вершины!" << endl;
deleteGraph(graph, numG, nullptr, nullptr);
return 1;
}
DFSDistance(graph, numG, startVertex);
deleteGraph(graph, numG, nullptr, nullptr);
return 0;
}
```

Файл lab_9_3_3.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>
#include <chrono>
#include <iomanip>
#include <vector>

using namespace std;
using namespace chrono;

struct ListNode {
int vertex;
ListNode* next;
ListNode(int v) : vertex(v), next(nullptr) {}
};

struct AdjacencyList {
ListNode* head;
int size;
AdjacencyList() : head(nullptr), size(0) {}
void add(int v) {
ListNode* newNode = new ListNode(v);
newNode->next = head;
head = newNode;
size++;
}
~AdjacencyList() {
ListNode* current = head;
while (current != nullptr) {
ListNode* next = current->next;
delete current;
current = next;
}
head = nullptr;
size = 0;
}
};
```

```
AdjacencyList* createGraph(int n) {
AdjacencyList* graph = new AdjacencyList[n];
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (rand() % 2 == 1) {
graph[i].add(j);
graph[j].add(i);
}
}
}
return graph;
}

void deleteGraph(AdjacencyList* graph, int n) {
delete[] graph;
}

void DFSDRecursive(AdjacencyList* graph, int numG, int* visited, int* distance, int current, int
currentDistance) {
visited[current] = 1;
distance[current] = currentDistance;
ListNode* neighbor = graph[current].head;
while (neighbor != nullptr) {
int neighborVertex = neighbor->vertex;
if (visited[neighborVertex] == 0) {
DFSDRecursive(graph, numG, visited, distance, neighborVertex, currentDistance + 1);
}
neighbor = neighbor->next;
}
}

void DFSDistance(AdjacencyList* graph, int numG, int startVertex, int* distance) {
int* visited = new int[numG];
for (int i = 0; i < numG; i++) {
visited[i] = 0;
distance[i] = -1;
}
DFSDRecursive(graph, numG, visited, distance, startVertex, 0);
delete[] visited;
}

void BFSD(AdjacencyList* graph, int numG, int startVertex, int* distance) {
queue<int> q;
for (int i = 0; i < numG; i++) {
distance[i] = -1;
}
distance[startVertex] = 0;
q.push(startVertex);

while (!q.empty()) {
int v = q.front();
```

```
q.pop();

ListNode* current = graph[v].head;
while (current != nullptr) {
int neighbor = current->vertex;
if (distance[neighbor] == -1) {
q.push(neighbor);
distance[neighbor] = distance[v] + 1;
}
current = current->next;
}
}
}

struct TestResult {
int vertices;
string algorithm;
long long timeMicroseconds;
double timeMilliseconds;
};

TestResult runTest(AdjacencyList* graph, int numVertices, int startVertex,
const string& algorithm, int* distDFS, int* distBFS) {
TestResult result;
result.vertices = numVertices;
result.algorithm = algorithm;
auto start = high_resolution_clock::now();
if (algorithm == "DFS") {
DFSDistance(graph, numVertices, startVertex, distDFS);
} else {
BFSD(graph, numVertices, startVertex, distBFS);
}
auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start);
result.timeMicroseconds = duration.count();
result.timeMilliseconds = duration.count() / 1000.0;
return result;
}

void printTable(const vector<TestResult>& results) {
cout << "================================================================" <<
endl;
cout << "СРАВНЕНИЕ ВРЕМЕНИ ВЫПОЛНЕНИЯ DFS И BFS" << endl;
cout << "================================================================" <<
endl;
cout << left << setw(12) << " Вершин "
<< left << setw(12) << " Алгоритм "
<< left << setw(20) << " Время (микросек) "
<< left << setw(20) << " Время (миллисек) " << endl;
cout << "----------------------------------------------------------------" << endl;
for (const auto& result : results) {
```

```
cout << left << setw(12) << result.vertices
<< left << setw(12) << result.algorithm
<< left << setw(20) << result.timeMicroseconds
<< fixed << setprecision(3) << left << setw(20) << result.timeMilliseconds << endl;
}
cout << "=========================================================================" <<
endl;
}

void printComparison(const vector<TestResult>& results) {
cout << "\n=========================================================================" <<
endl;
cout << "СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ" << endl;
cout << "=========================================================================" <<
endl;
cout << left << setw(12) << " Вершин "
<< left << setw(15) << " Быстрее "
<< left << setw(20) << " Разница (микросек) "
<< left << setw(15) << " Выигрыш (%) " << endl;
cout << "-------------------------------------------------------------------------" << endl;
for (int i = 0; i < results.size(); i += 2) {
if (i + 1 < results.size()) {
const TestResult& dfs = results[i];
const TestResult& bfs = results[i + 1];
string faster;
long long diff;
double percentage;
if (dfs.timeMicroseconds < bfs.timeMicroseconds) {
faster = "DFS";
diff = bfs.timeMicroseconds - dfs.timeMicroseconds;
percentage = (double)diff / bfs.timeMicroseconds * 100;
} else {
faster = "BFS";
diff = dfs.timeMicroseconds - bfs.timeMicroseconds;
percentage = (double)diff / dfs.timeMicroseconds * 100;
}
cout << left << setw(12) << dfs.vertices
<< left << setw(15) << faster
<< left << setw(20) << diff
<< fixed << setprecision(2) << left << setw(15) << percentage << "%" << endl;
}
}
cout << "=========================================================================" <<
endl;
}

int main() {
srand(time(0));
vector<int> graphSizes = {100, 400, 800, 2000, 5000, 10000};
const int startVertex = 0;
vector<TestResult> allResults;
```

```
cout << "Запуск тестов производительности..." << endl;
cout << "Размеры графов: ";
for (int size : graphSizes) {
cout << size << " ";
}
cout << "\n\n";
for (int numVertices : graphSizes) {
cout << "Тестирование графа с " << numVertices << " вершинами..." << endl;
AdjacencyList* graph = createGraph(numVertices);
int* distDFS = new int[numVertices];
int* distBFS = new int[numVertices];
TestResult dfsResult = runTest(graph, numVertices, startVertex, "DFS", distDFS, distBFS);
allResults.push_back(dfsResult);
TestResult bfsResult = runTest(graph, numVertices, startVertex, "BFS", distDFS, distBFS);
allResults.push_back(bfsResult);
bool resultsMatch = true;
for (int i = 0; i < numVertices; i++) {
if (distDFS[i] != distBFS[i]) {
resultsMatch = false;
break;
}
}
if (!resultsMatch) {
cout << " ВНИМАНИЕ: результаты DFS и BFS не совпадают!" << endl;
} else {
cout << " √ Результаты DFS и BFS совпадают" << endl;
}
int totalEdges = 0;
for (int i = 0; i < numVertices; i++) {
ListNode* current = graph[i].head;
while (current != nullptr) {
totalEdges++;
current = current->next;
}
}
totalEdges /= 2;
cout << " Ребра: " << totalEdges
<< ", Плотность: " << fixed << setprecision(2)
<< (2.0 * totalEdges) / (numVertices * (numVertices - 1)) * 100 << "%" << endl;
delete[] distDFS;
delete[] distBFS;
deleteGraph(graph, numVertices);
cout << endl;
}
printTable(allResults);
printComparison(allResults);
cout << "\n================================================================" <<
endl;
cout << "СТАТИСТИЧЕСКИЙ АНАЛИЗ" << endl;
cout << "================================================================" <<
endl;
```

```
double totalDFSTime = 0;
double totalBFSTime = 0;
int dfsWins = 0;
int bfsWins = 0;
for (int i = 0; i < allResults.size(); i += 2) {
if (i + 1 < allResults.size()) {
const TestResult& dfs = allResults[i];
const TestResult& bfs = allResults[i + 1];
totalDFSTime += dfs.timeMilliseconds;
totalBFSTime += bfs.timeMilliseconds;
if (dfs.timeMicroseconds < bfs.timeMicroseconds) {
dfsWins++;
} else {
bfsWins++;
}
}
}
cout << "Среднее время DFS: " << fixed << setprecision(3)
<< totalDFSTime / (allResults.size() / 2) << " мс" << endl;
cout << "Среднее время BFS: " << totalBFSTime / (allResults.size() / 2) << " мс" << endl;
cout << "DFS был быстрее в " << dfsWins << " из " << (allResults.size() / 2) << " тестов" << endl;
cout << "BFS был быстрее в " << bfsWins << " из " << (allResults.size() / 2) << " тестов" << endl;
return 0;
}
```