Министерство образования и науки Российской Федерации

ФГБОУ ВО «Пензенский государственный университет»

Факультет вычислительной техники

Кафедра «Вычислительная техника»

Пенза 2025

**Отчет**

по лабораторной работе №9

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

Выполнил: студент группы 24ВВВ1

Егорова К.А.

Винникова Е.С

Марушкин Д.И.

Проверил: к.т.н., доцент

Юрова О.В.

Деев М.В.

**Лабораторные задания:**

### **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 С++.

**3.**\* Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

### **Задание 2\***

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.
2. Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.
3. Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

### **Пояснительный текст к программе**

#### **1. Назначение программы**

Цель программы — изучение алгоритмов обхода графа и поиска расстояний между вершинами, а также сравнение работы алгоритмов при разных способах представления графа.

Программа позволяет:

* генерировать случайный **неориентированный граф** заданного размера;
* выводить граф на экран в виде **матрицы смежности** и **списков смежности**;
* выполнять обход графа и вычислять **расстояние от стартовой вершины до всех остальных** четырьмя способами:
  1. DFS по матрице смежности (стек std::stack);
  2. BFS по матрице смежности (очередь std::queue);
  3. DFS по спискам смежности;
  4. BFS по спискам смежности;
* измерять и выводить **время выполнения каждого алгоритма** с помощью std::chrono.

#### **2. Структура программы**

**Основной класс — Graph**:

* matrix — матрица смежности графа (vertexCount × vertexCount).
* adjList — списки смежности графа (vector<vector<int>>).
* visited — массив посещённых вершин (для обхода DFS).
* vertexCount — количество вершин.
* edgeCount — количество рёбер.

**Основные методы класса**:

1. genAdjMatrix(double probability) — генерация матрицы смежности случайного графа с вероятностью появления ребра probability. Метод также формирует корректный список смежности (adjList) после генерации матрицы.
2. genAdjList() — отдельная процедура формирования списков смежности на основе матрицы смежности.
3. print() — вывод матрицы и списка смежности на экран, включая количество вершин и рёбер.
4. BFS(int start) — обход в ширину по матрице смежности, подсчёт расстояний до каждой вершины.
5. BFS\_adj(int start) — обход в ширину по спискам смежности, подсчёт расстояний.
6. DFS(int start) — обход в глубину по матрице смежности, подсчёт расстояний.
7. DFS\_adj(int start) — обход в глубину по спискам смежности, подсчёт расстояний.

**Вспомогательные методы**:

* resetVisited() — сброс флагов посещённых вершин перед обходом DFS.

#### **3. Функция измерения времени**

void TimeToTime(void (Graph::\* func)(int), Graph& obj, int param)

Позволяет измерять время выполнения любого метода класса Graph. Использует std::chrono::high\_resolution\_clock для точного измерения микросекунд.

#### **4. Основная функция main()**

1. Создаёт граф Graph gp(10) с 10 вершинами.
2. Генерирует случайную матрицу смежности через genAdjMatrix().
3. Выводит граф на экран с помощью print().
4. Выполняет один из алгоритмов обхода (в текущем коде — DFS по матрице) и выводит **расстояния до всех вершин**.
5. (При желании можно добавлять вызовы всех остальных методов и измерять их время через TimeToTime.)

#### **5. Используемые средства**

* **Язык:** C++17
* **Стандартные библиотеки:**
  + <iostream> — ввод/вывод
  + <vector> — хранение матрицы и списков смежности
  + <queue> — обход BFS
  + <stack> — обход DFS
  + <random> — генерация случайных чисел
  + <chrono> — измерение времени выполнения
  + <iomanip> — форматированный вывод
* **Алгоритмы:** BFS и DFS для невзвешенного графа.
* **Принципы работы BFS и DFS:**
  + BFS — использует очередь FIFO, гарантирует поиск кратчайшего пути (по числу рёбер) от стартовой вершины;
  + DFS — использует стек LIFO, позволяет обойти граф глубоко, расстояние считается в порядке обхода.

**6. Результат работы программы:**

Для каждого графа выполняются оба варианта обхода в ширину и в глубину, начиная с вершины 0, и фиксируется время выполнения.

Для каждого размера графа (число вершин n = 1000, 2000, 4000, 8000, 16000) с фиксированной вероятностью связи создаётся отдельный граф.

Таблица 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | DFS мастрица мс | BFS матрица мс | DFS\_список мс | BFS\_список мс |
| 100 | 110 | 1 | 64 | 1 |
| 200 | 1010 | 14 | 497 | 1 |
| 400 | 2622 | 5 | 2156 | 1 |
| 800 | 19110 | 15 | 16992 | 5 |
| 1600 | 157530 | 52 | 170573 | 13 |

Диаграмма 1.

### **5. Анализ и вывод**

Анализ результатов показал, что время выполнения DFS значительно растёт с увеличением числа вершин и рёбер. При использовании матрицы смежности DFS работает дольше, чем при списке смежности, особенно на небольших графах. При больших графах разница сохраняется, но в целом DFS остаётся значительно более затратным по времени, чем BFS.

Время выполнения BFS практически не зависит от числа вершин и способа хранения графа. Оно остаётся стабильным во всех экспериментах. Исходя из этого, DFS чувствителен к размеру графа и способу его хранения, тогда как BFS демонстрирует стабильное время выполнения независимо от структуры графа.

**Трассировка программы:**

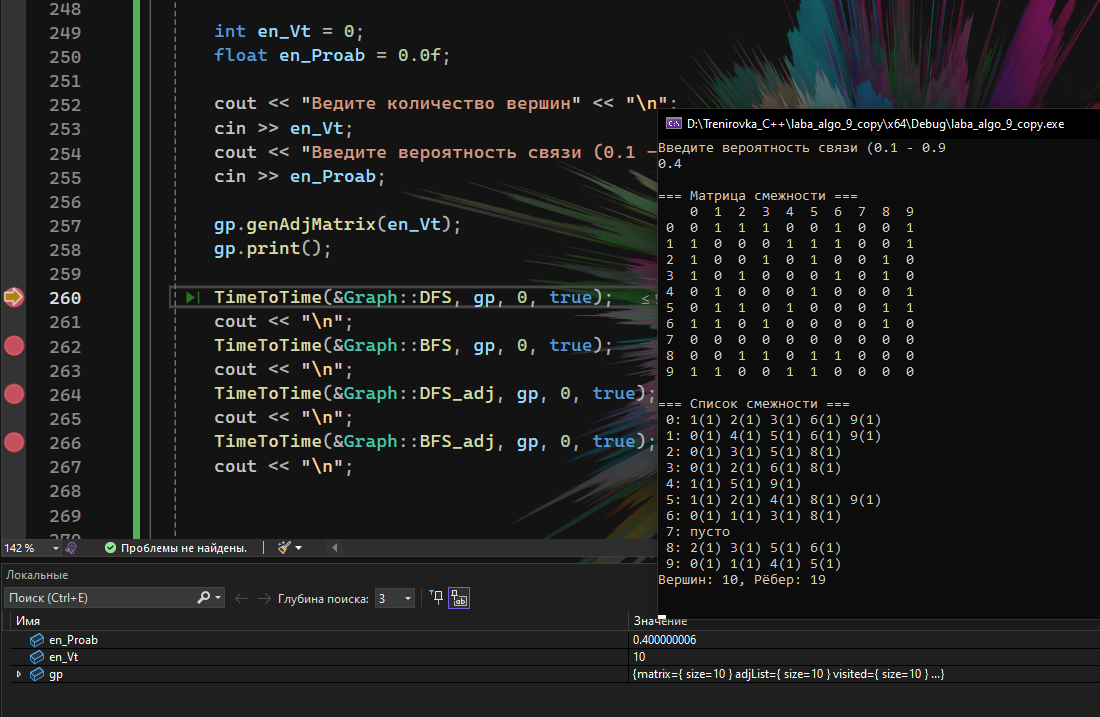


Рисунок 1. Ввод значений и создание графа.

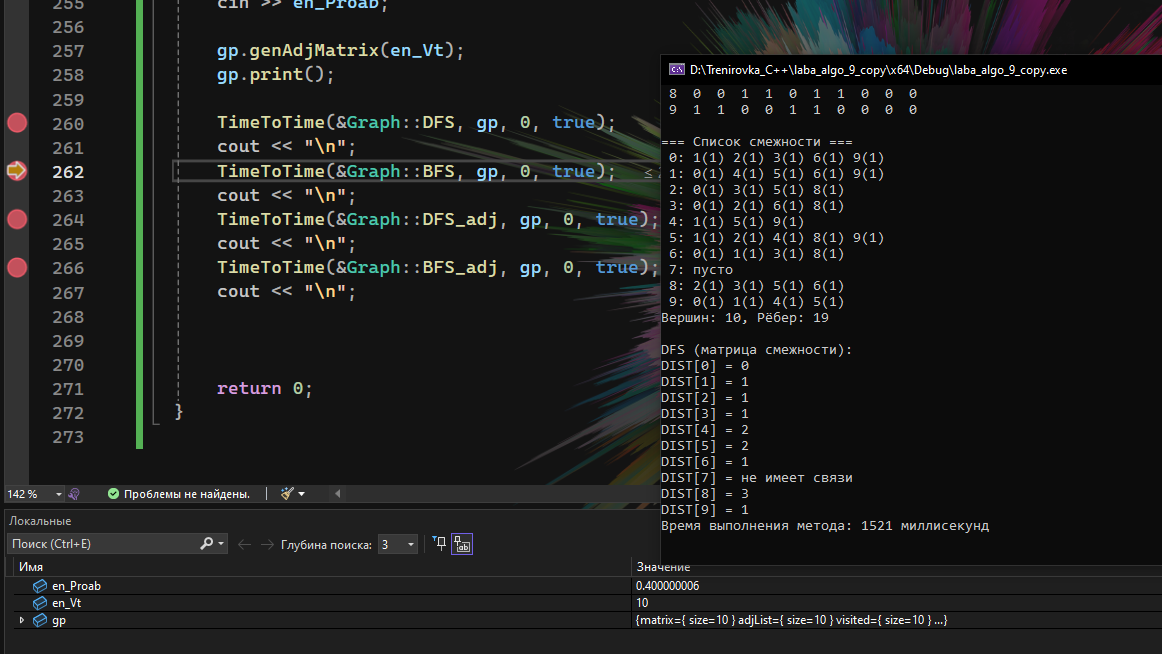


Рисунок 2 – Поиск расстояния DFS по матрице смежности.

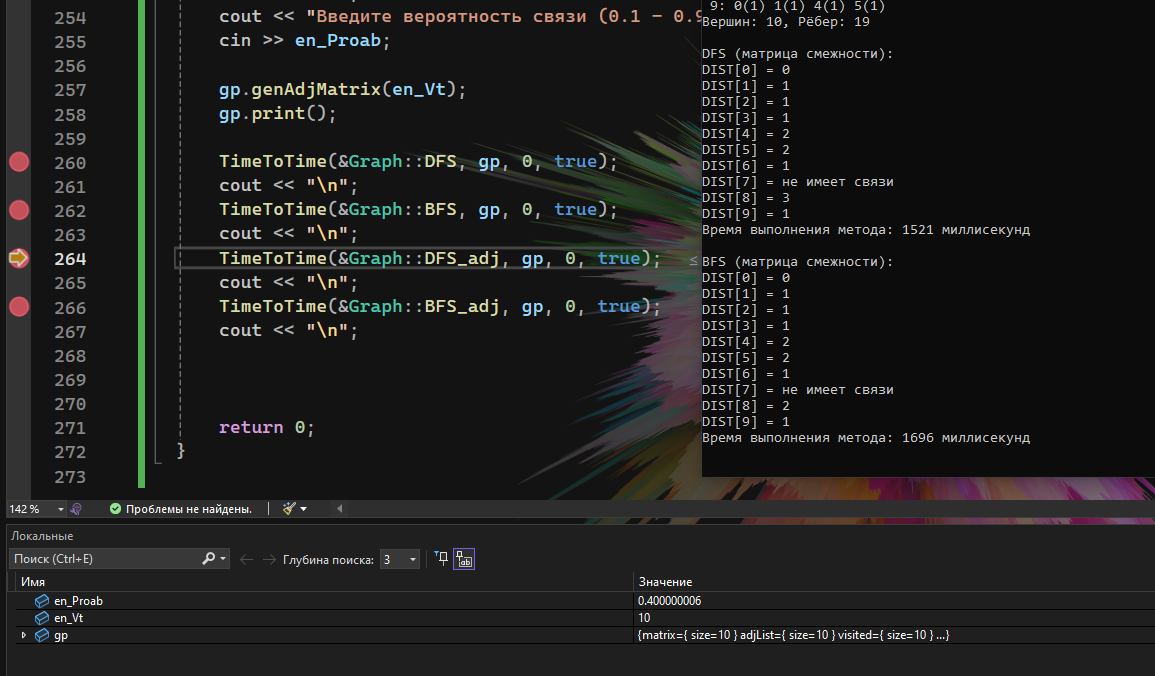


Рисунок 2 – Поиск расстояния BFS по матрице смежности.

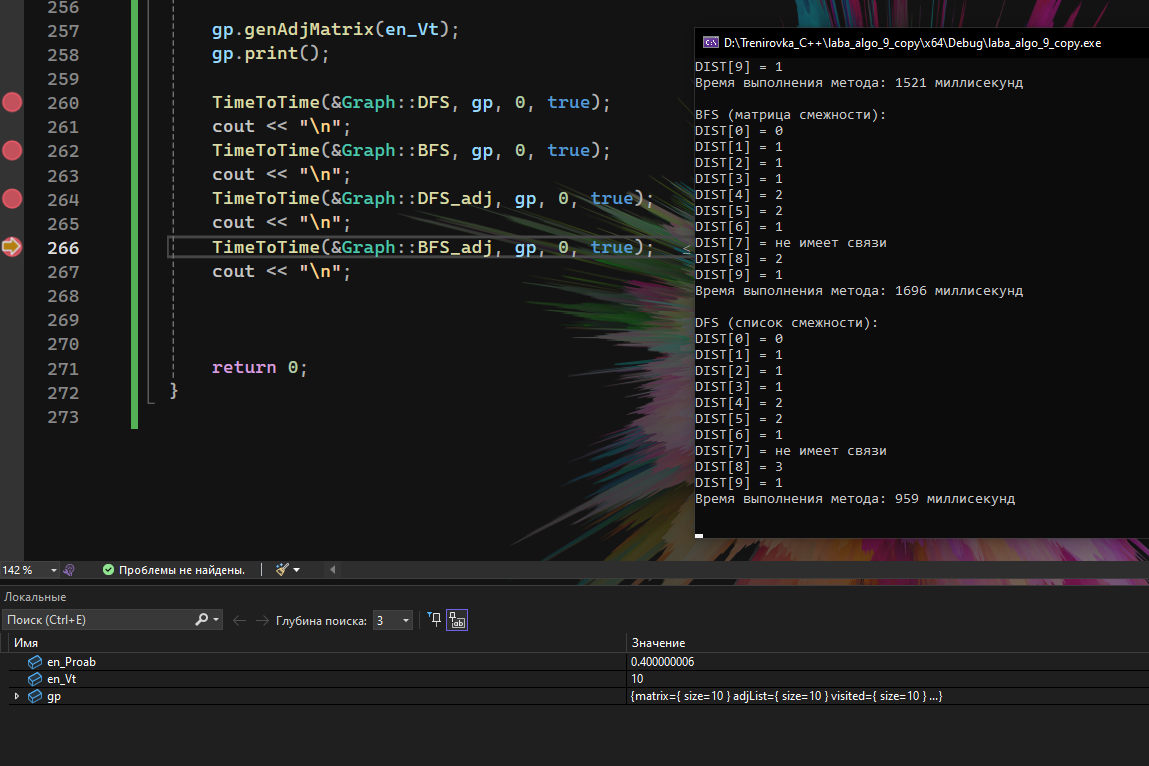


Рисунок 3 - Поиск расстояния DFS по матрице смежности.

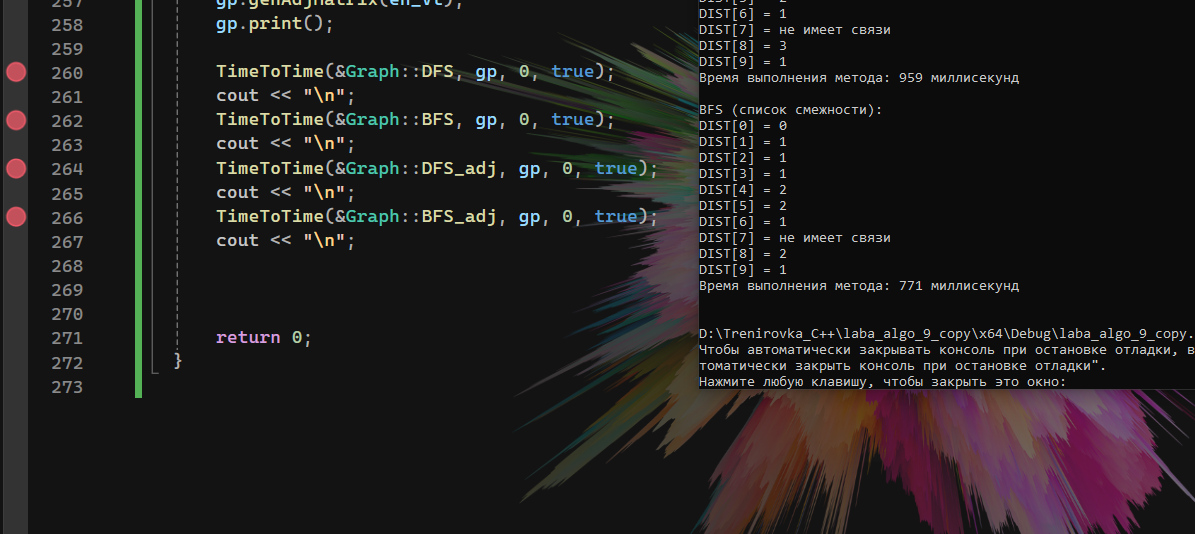


Рисунок 4 - Поиск расстояния BFS по списку смежности.

**Результат работы программы:**

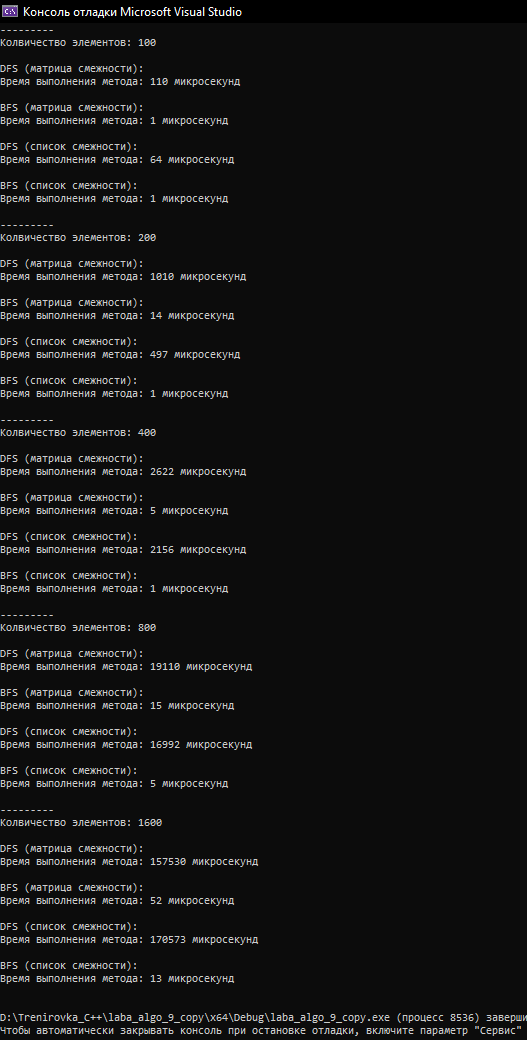


Рисунок 10 – Результат работы программы.

**Листинг программы:**

#include <iostream>

#include <vector>

#include <queue>

#include <chrono>

#include <random>

#include <stack>

#include <string>

#include <iomanip>

using namespace std;

using namespace std::chrono;

class Graph

{

public:

Graph() : vertexCount(0), edgeCount(0)

{

}

void genAdjMatrix(int vCount, double probability = 0.4)

{

matrix.assign(vCount, vector<int>(vCount, 0));

adjList.assign(vCount, {});

visited.assign(vCount, false);

random\_device rd;

mt19937 gen(rd());

uniform\_real\_distribution<> dist(0.0, 1.0);

vertexCount = vCount;

// Генерация ребер

for (int i = 0; i < vertexCount; i++)

{

for (int j = i + 1; j < vertexCount; j++)

{

if (dist(gen) < probability)

{

matrix[i][j] = matrix[j][i] = 1;

++edgeCount;

}

}

}

genAdjList();

}

void DFS(int start, bool show)

{

resetVisited();

stack<int> st;

st.push(start);

vector<int> dist(vertexCount, -1);

dist[start] = 0;

while (!st.empty())

{

int v = st.top();

st.pop();

if (!visited[v])

{

visited[v] = true;

}

for (int i = vertexCount - 1; i >= 0; i--)

{

if (matrix[v][i] == 1 && !visited[i])

{

if (dist[i] == -1)

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

st.push(i);

}

}

}

cout << "DFS (матрица смежности):\n";

if (show)

showDist(dist);

}

void BFS(int start, bool show)

{

vector<int> dist(vertexCount, -1);

queue<int> q;

dist[start] = 0;

q.push(start);

while (!q.empty())

{

int v = q.front();

q.pop();

for (int i = 0; i < vertexCount; i++)

{

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

{

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

q.push(i);

}

}

}

cout << "BFS (матрица смежности):\n";

if(show)

showDist(dist);

}

void DFS\_adj(int start, bool show)

{

resetVisited();

stack<int> st;

st.push(start);

vector<int> dist(vertexCount, -1);

dist[start] = 0;

while (!st.empty())

{

int v = st.top();

st.pop();

if (!visited[v])

{

visited[v] = true;

}

for (int nb : adjList[v])

{

if (!visited[nb])

{

if (dist[nb] == -1)

dist[nb] = dist[v] + 1;

st.push(nb);

}

}

}

cout << "DFS (список смежности):\n";

if (show)

showDist(dist);

}

void BFS\_adj(int start, bool show)

{

vector<int> dist(vertexCount, -1);

queue<int> q;

dist[start] = 0;

q.push(start);

while (!q.empty())

{

int v = q.front();

q.pop();

for (int nb : adjList[v])

{

if (dist[nb] == -1)

{

dist[nb] = dist[v] + 1;

q.push(nb);

}

}

}

cout << "BFS (список смежности):\n";

if (show)

showDist(dist);

}

void print() const {

cout << "\n=== Матрица смежности ===\n ";

for (int j = 0; j < vertexCount; j++) cout << setw(2) << j << " ";

cout << "\n";

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

cout << setw(2) << i << " ";

for (int j = 0; j < vertexCount; j++)

cout << setw(2) << matrix[i][j] << " ";

cout << "\n";

}

cout << "\n=== Список смежности ===\n";

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

cout << setw(2) << i << ": ";

if (adjList[i].empty()) cout << "пусто";

else for (int j : adjList[i]) cout << j << "(" << matrix[i][j] << ") ";

cout << "\n";

}

cout << "Вершин: " << vertexCount << ", Рёбер: " << edgeCount << "\n\n";

}

private:

void resetVisited()

{

fill(visited.begin(), visited.end(), false);

}

void genAdjList()

{

for (int i = 0; i < vertexCount; i++)

{

adjList[i].clear();

for (int j = 0; j < vertexCount; j++)

{

if (matrix[i][j])

adjList[i].push\_back(j);

}

}

}

void showDist(vector<int>& dist)

{

for (int i = 0; i < vertexCount; i++)

cout << "DIST[" << i << "] = " << (dist[i] == -1 ? "не имеет связи" : std::to\_string(dist[i])) << "\n";

}

vector<vector<int>> matrix;

vector<vector<int>> adjList;

vector<bool> visited;

int vertexCount;

int edgeCount;

};

// Обёртка для метода класса

void TimeToTime(void (Graph::\* func)(int, bool), Graph& obj, int param, bool show)

{

auto start = high\_resolution\_clock::now();

(obj.\*func)(param, show); // вызываем метод

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << "Время выполнения метода: " << duration.count() << " миллисекунд\n";

}

void testTime()

{

std::vector<int> capacity{100, 200, 400, 800, 1600};

for (int i = 0; i < capacity.size(); i++)

{

cout << setfill('-') << setw(10) << '\n';

cout << "Колвичество элементов: " << capacity[i] << "\n\n";

Graph gp;

gp.genAdjMatrix(capacity[i]);

TimeToTime(&Graph::DFS, gp, 0, false);

cout << "\n";

TimeToTime(&Graph::BFS, gp, 0, false);

cout << "\n";

TimeToTime(&Graph::DFS\_adj, gp, 0, false);

cout << "\n";

TimeToTime(&Graph::BFS\_adj, gp, 0, false);

cout << "\n";

}

}

int main()

{

setlocale(LC\_ALL, "RUSSIAN");

Graph gp;

int en\_Vt = 0;

float en\_Proab = 0.0f;

cout << "Ведите количество вершин" << "\n";

cin >> en\_Vt;

cout << "Введите вероятность связи (0.1 - 0.9" << "\n";

cin >> en\_Proab;

gp.genAdjMatrix(en\_Vt);

gp.print();

TimeToTime(&Graph::DFS, gp, 0, true);

cout << "\n";

TimeToTime(&Graph::BFS, gp, 0, true);

cout << "\n";

TimeToTime(&Graph::DFS\_adj, gp, 0, true);

cout << "\n";

TimeToTime(&Graph::BFS\_adj, gp, 0, true);

cout << "\n";

return 0;

}