Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# Описание задачи.

Целью лабораторной работы была реализация генератора случайных графов на языке C++. Генератор должен был поддерживать следующие параметры:

* Максимальное и минимальное количество вершин.
* Максимальное и минимальное количество ребер.
* Максимальное количество ребер, связанных с одной вершиной.
* Указание, является ли граф направленным.
* Максимальное количество входящих и выходящих ребер для направленных графов.

**Задание:**

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

* Максимальное/Минимальное количество генерируемых вершин
* Максимальное/Минимальное количество генерируемых ребер
* Максимальное количество ребер связанных с одной вершины
* Генерируется ли направленный граф
* Максимальное количество входящих и выходящих ребер

Сгенерированный граф должен быть описан в рамках одного класса(этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:

* Выдача матрицы смежности
* Выдача матрицы инцидентности
* Выдача список смежности
* Выдача списка ребер

В качестве проверки работоспособности, требуется сгенерировать 10 графов с возрастающим количеством вершин и ребер(количество выбирать в зависимости от сложности расчета для вашего отдельно взятого ПК). На каждом из сгенерированных графов требуется выполнить поиск кратчайшего пути или подтвердить его отсутствие из точки А в точку Б, выбирающиеся случайным образом заранее, поиском в ширину и поиском в глубину, замерев время требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

# Описание метода/модели.

Граф — это структура данных, состоящая из вершин (узлов) и ребер (связей между вершинами). В данной работе граф может быть как направленным, так и ненаправленным.

Методы представления графа:

1. Матрица смежности — двумерный массив, где элемент matrix[i][j] указывает на наличие ребра между вершинами i и j.
2. Матрица инцидентности — матрица, где строки соответствуют вершинам, а столбцы — ребрам. Элемент matrix[i][j] указывает на связь вершины i с ребром j.
3. Список смежности — массив списков, где каждый элемент массива соответствует вершине, а список содержит соседей этой вершины.
4. Список ребер — массив пар, где каждая пара представляет собой ребро между двумя вершинами.

Алгоритмы поиска:

1. Поиск в ширину (BFS) — алгоритм, который обходит граф "по уровням", начиная с заданной вершины. Он эффективен для поиска кратчайшего пути в ненаправленных графах.
2. Поиск в глубину (DFS) — алгоритм, который обходит граф "вглубь", исследуя каждую ветвь до конца. Он менее эффективен для поиска кратчайшего пути, но может быть полезен для анализа структуры графа.

**Для чего используются**

* **BFS**:
  + Поиск кратчайшего пути в невзвешенных графах.
  + Проверка, является ли граф двудольным.
  + Обход веб-страниц уровнями.
* **DFS**:
  + Обнаружение циклов в графе.
  + Поиск связных компонентов.
  + Топологическая сортировка в ациклических направленных графах.
  + Решение головоломок, таких как лабиринты.

**Примеры**

* **BFS**: Найти кратчайший путь от A до C в графе A-B-C: BFS посетит A, затем B, затем C, найдя путь длиной 2.
* **DFS**: Проверить, есть ли цикл в графе A-B-C-A: DFS может пройти A-B-C, затем вернуться к A, обнаружив цикл.

**Подробное описание BFS**

**Как работает BFS:**

* BFS начинается с исходной вершины и исследует все её соседи на текущем уровне, прежде чем перейти к следующему уровню. Это достигается с помощью очереди (FIFO, First-In-First-Out), где вершины добавляются в конец и извлекаются из начала.
* Алгоритм:
  1. Начните с исходной вершины, добавьте её в очередь и отметьте как посещенную.
  2. Пока очередь не пуста:
     + Извлеките вершину из начала очереди.
     + Для каждого не посещенного соседа:
       - Добавьте его в очередь.
       - Отметьте как посещенную.
       - Опционально, запишите путь или расстояние от исходной вершины.
* Пример: Рассмотрим граф A-B-C, где A связано с B, B с C. Начав с A, BFS посетит A, затем B (сосед A), затем C (сосед B), что даёт порядок A, B, C.

**Использование BFS:**

* **Кратчайший путь в невзвешенных графах**: BFS гарантирует, что найденный путь минимален по количеству рёбер, так как исследует уровни последовательно. Например, в графе A-B-C, путь A-C через B будет найден как кратчайший (длина 2).
* **Проверка двудольности**: BFS может использоваться для раскраски графа уровнями, чтобы проверить, можно ли разделить вершины на две группы без рёбер внутри групп.
* **Веб-кравлеры**: Используется для обхода веб-страниц уровнями, начиная с главной страницы и исследуя все ссылки на текущем уровне, прежде чем перейти к следующим.
* Исследования показывают, что BFS эффективен для задач, где важна близость к исходной вершине, например, в социальных сетях для поиска друзей в радиусе двух шагов.

**Пример:**

* Граф: A-B, A-D, B-C, D-E. Начав с A, BFS посетит: A, затем B и D, затем C и E, что даёт порядок A, B, D, C, E.

**Подробное описание DFS**

**Как работает DFS:**

* DFS углубляется по одной ветви, пока не достигнет конца, затем возвращается назад, исследуя другие ветви. Использует стек (LIFO, Last-In-First-Out) или рекурсию для отслеживания вершин.
* Алгоритм:
  1. Начните с исходной вершины, добавьте её в стек и отметьте как посещенную.
  2. Пока стек не пуст:
     + Извлеките вершину из вершины стека.
     + Для каждого не посещенного соседа:
       - Добавьте его в стек.
       - Отметьте как посещенную.
       - Опционально, запишите путь.
* Пример: В том же графе A-B-C, начав с A, DFS может посетить A, затем B, затем C, затем вернуться к B, затем к A, и так далее, в зависимости от порядка выбора соседей.

**Использование DFS:**

* **Обнаружение циклов**: DFS может обнаружить цикл, если вершина уже посещена и не является родителем текущего пути, что полезно для проверки ацикличности графа.
* **Поиск связных компонентов**: DFS может использоваться для идентификации всех вершин, достижимых из одной вершины, что помогает найти компоненты в несвязных графах.
* **Топологическая сортировка**: В направленных ациклических графах (DAG) DFS может использоваться для упорядочивания вершин, где для каждой вершины все её предшественники идут раньше.
* **Решение головоломок**: Например, в лабиринтах DFS может найти путь от входа к выходу, исследуя одну ветвь до конца, что эффективно для задач с одним решением.

**Пример:**

* Граф: A-B, A-D, B-C, D-E. Начав с A, DFS может посетить A, затем B, затем C, вернуться к B, затем к A, затем D, затем E, что даёт порядок A, B, C, D, E, в зависимости от выбора.

**Сравнение производительности**

Оба алгоритма имеют временную сложность O(V + E), но их реальное время выполнения зависит от структуры графа:

* BFS требует больше памяти для широких графов, так как очередь может содержать все вершины на текущем уровне.
* DFS требует меньше памяти для глубоких графов, так как стек хранит только путь до текущей вершины.

В вашем эксперименте, где DFS показал лучшую производительность для больших графов, это может быть связано с тем, что сгенерированные графы имели глубокие пути, где DFS мог быстрее найти вершину B, в то время как BFS исследовал все уровни, что увеличивало время. Это подтверждается данными графика, где на 900 вершинах BFS занял около 1.4 секунд, а DFS — менее 0.3 секунд.

**Таблица сравнения**

| **Характеристика** | **BFS** | **DFS** |
| --- | --- | --- |
| Структура данных | Очередь (FIFO) | Стек (LIFO) или рекурсия |
| Подход | Уровневый, ширина | Глубокий, ветвь за ветвью |
| Память | Высокая для широких графов | Низкая для глубоких графов |
| Применение | Кратчайший путь, двудольность | Циклы, топологическая сортировка |
| Пример использования | Веб-кравлеры, социальные сети | Лабиринты, DAG сортировка |

# 

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <random>

#include <chrono>

#include <unordered\_set>

#include <utility>

#include <algorithm>

#include <iomanip>

#include <fstream>

using namespace std;

class Graph {

private:

int vertices;

vector<vector<int>> adjacencyMatrix;

vector<vector<int>> incidenceMatrix;

vector<vector<int>> adjacencyList;

vector<pair<int, int>> edgeList;

bool directed;

vector<int> inDegree; // Для отслеживания входящих рёбер

vector<int> outDegree; // Для отслеживания исходящих рёбер

public:

Graph(int v, bool dir) : vertices(v), directed(dir) {

// Инициализируем матрицу смежности размером vertices x vertices, заполненную нулями

adjacencyMatrix.resize(vertices, vector<int>(vertices, 0));

// Инициализируем список смежности с количеством вершин

adjacencyList.resize(vertices);

//Инициализируем векторы для входящих и исходящих рёбер, заполняя их нулями

inDegree.resize(vertices, 0);

outDegree.resize(vertices, 0);

}

// Добавление рёбер в граф с проверками на максимальное количество рёбер для каждой вершины

void addEdge(int u, int v, int maxDegree) {

//Максимальное количество ребер, связанных с одной вершиной

//количество исходящих рёбер у вершины u и входящих рёбер у вершины v не превышает заданного значения maxDegree

if (outDegree[u] < maxDegree && inDegree[v] < maxDegree) { // Проверка максимальной степени

adjacencyMatrix[u][v] = 1; // Устанавливаем ребро в матрице смежности

adjacencyList[u].push\_back(v); // Добавляем v в список смежности для u

edgeList.emplace\_back(u, v); // Добавляем пару (u, v) в список рёбер

outDegree[u]++; // Увеличиваем количество исходящих рёбер для u

inDegree[v]++; // Увеличиваем количество входящих рёбер для v

if (!directed) { // Для неориентированного графа добавляем обратное ребро

adjacencyMatrix[v][u] = 1;

adjacencyList[v].push\_back(u);

outDegree[v]++;

inDegree[u]++;

}

}

}

// Метод для получения матрицы смежности

vector<vector<int>> getAdjacencyMatrix() const {

return adjacencyMatrix;

}

// Метод для получения матрицы инцидентности

vector<vector<int>> getIncidenceMatrix() const {

vector<vector<int>> incidenceMatrix(vertices, vector<int>(edgeList.size(), 0));

for (size\_t i = 0; i < edgeList.size(); ++i) {

int u = edgeList[i].first;

int v = edgeList[i].second;

incidenceMatrix[u][i] = 1;

if (!directed) {

incidenceMatrix[v][i] = -1;

} else {

incidenceMatrix[v][i] = 1;

}

}

return incidenceMatrix;

}

// Метод для получения списка смежности

vector<vector<int>> getAdjacencyList() const {

return adjacencyList;

}

// Метод для получения списка рёбер

vector<pair<int, int>> getEdgeList() const {

return edgeList;

}

// Метод для выполнения поиска в ширину (BFS)

vector<int> BFS(int start, int end) {

if (start < 0 || start >= vertices || end < 0 || end >= vertices) {

return {};

}

vector<bool> visited(vertices, false);

queue<int> q;

vector<int> parent(vertices, -1);

q.push(start);

visited[start] = true;

while (!q.empty()) {

int node = q.front(); q.pop();

if (node == end) break;

for (int neighbor : adjacencyList[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

parent[neighbor] = node;

q.push(neighbor);

}

}

}

vector<int> path;

if (parent[end] != -1) {

for (int v = end; v != -1; v = parent[v]) path.push\_back(v);

reverse(path.begin(), path.end());

}

return path;

}

// Метод для выполнения поиска в глубину (DFS)

vector<int> DFS(int start, int end) {

if (start < 0 || start >= vertices || end < 0 || end >= vertices) {

return {};

}

vector<bool> visited(vertices, false);

stack<int> s;

vector<int> parent(vertices, -1);

s.push(start);

visited[start] = true;

while (!s.empty()) {

int node = s.top(); s.pop();

if (node == end) break;

for (int neighbor : adjacencyList[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

parent[neighbor] = node;

s.push(neighbor);

}

}

}

vector<int> path;

if (parent[end] != -1) {

for (int v = end; v != -1; v = parent[v]) path.push\_back(v);

reverse(path.begin(), path.end());

}

return path;

}

};

int main() {

random\_device rd;

mt19937 gen(rd());

vector<double> bfsTimes, dfsTimes;

vector<int> sizes;

// Открытие файла для записи результатов

ofstream outputFile("graph\_results.csv");

outputFile << "Graph,Vertices,Edges,BFS Time (s),DFS Time (s)\n";

for (int i = 1; i <= 10; ++i) {

// Диапазон вершин и рёбер увеличивается с каждой итерацией

int minVertices = 10 + (i - 1) \* 10;

int maxVertices = 10 + i \* 10;

int minEdges = 20 + (i - 1) \* 20;

int maxEdges = 20 + i \* 20;

// // Генерация случайного количества вершин, рёбер, максимальной степени и флага направленности

uniform\_int\_distribution<int> vertexDist(minVertices, maxVertices);

uniform\_int\_distribution<int> edgeDist(minEdges, maxEdges);

uniform\_int\_distribution<int> degreeDist(3, 10);

uniform\_int\_distribution<int> directedDist(0, 1);

int vertices = vertexDist(gen); // Случайное количество вершин в диапазоне

int edges = edgeDist(gen); // Случайное количество рёбер в диапазоне

int maxDegree = degreeDist(gen); // Максимальное количество рёбер на вершину

bool directed = directedDist(gen);

sizes.push\_back(vertices); // Добавляем размер графа в вектор

Graph graph(vertices, directed); // Создаём граф

// Генерация случайных рёбер

// в зависимости от значения переменной directed создаётся либо направленный граф (если directed равно true),

// либо неориентированный граф (если directed равно false).

uniform\_int\_distribution<int> vertexRange(0, vertices - 1);

int edgeCount = 0;

while (edgeCount < edges) {

int u = vertexRange(gen);

int v = vertexRange(gen);

if (u != v) {

graph.addEdge(u, v, maxDegree); // Добавление ребра с проверкой максимальной степени

edgeCount++;

}

}

int start = vertexRange(gen); // Случайная стартовая вершина

int end = vertexRange(gen); // Случайная конечная вершина

// Замер времени для поиска в ширину

auto startTime = chrono::high\_resolution\_clock::now();

auto bfsPath = graph.BFS(start, end);

auto endTime = chrono::high\_resolution\_clock::now();

double bfsTime = chrono::duration<double>(endTime - startTime).count();

bfsTimes.push\_back(bfsTime);

// Замер времени для поиска в глубину

startTime = chrono::high\_resolution\_clock::now();

auto dfsPath = graph.DFS(start, end);

endTime = chrono::high\_resolution\_clock::now();

double dfsTime = chrono::duration<double>(endTime - startTime).count();

dfsTimes.push\_back(dfsTime);

// Вывод результатов в консоль

cout << "Graph " << i << " (Vertices: " << vertices << ", Edges: " << edges << ")\n";

cout << "BFS Path: ";

for (int v : bfsPath) cout << v << " ";

cout << "\nBFS Time: " << bfsTime << " s\n";

cout << "DFS Path: ";

for (int v : dfsPath) cout << v << " ";

cout << "\nDFS Time: " << dfsTime << " s\n\n";

// Запись результатов в файл

outputFile << i << "," << vertices << "," << edges << "," << bfsTime << "," << dfsTime << "\n";

}

outputFile.close(); // Закрытие файла

// Визуализация данных

cout << "\nAnalysis of BFS vs DFS Times:\n";

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

cout << "Graph " << (i + 1) << " - BFS Time: " << bfsTimes[i] << " s, DFS Time: " << dfsTimes[i] << " s\n";

}

return 0;

}

**Результат:**

**C:\Users\user\CLionProjects\untitled15\cmake-build-debug\untitled15.exe**

**Graph 1 (Vertices: 17, Edges: 25)**

**BFS Path: 3 8 2 1 16**

**BFS Time: 6e-06 s**

**DFS Path: 3 8 2 1 16**

**DFS Time: 4.2e-06 s**

**Graph 2 (Vertices: 21, Edges: 49)**

**BFS Path: 19 12 9**

**BFS Time: 8.2e-06 s**

**DFS Path: 19 8 0 9**

**DFS Time: 8.9e-06 s**

**Graph 3 (Vertices: 30, Edges: 65)**

**BFS Path: 14 16 9 15 19 2**

**BFS Time: 9.1e-06 s**

**DFS Path: 14 26 23 4 28 11 17 2**

**DFS Time: 4.1e-06 s**

**Graph 4 (Vertices: 44, Edges: 87)**

**BFS Path: 23 28 10 20 8**

**BFS Time: 5.6e-06 s**

**DFS Path: 23 28 40 26 12 8**

**DFS Time: 3.2e-06 s**

**Graph 5 (Vertices: 50, Edges: 115)**

**BFS Path: 8 13 45 46 25**

**BFS Time: 9.8e-06 s**

**DFS Path: 8 6 21 14 10 25**

**DFS Time: 4.9e-06 s**

**Graph 6 (Vertices: 68, Edges: 136)**

**BFS Path:**

**BFS Time: 2.6e-06 s**

**DFS Path:**

**DFS Time: 1.6e-06 s**

**Graph 7 (Vertices: 76, Edges: 149)**

**BFS Path: 39 2 24 13 41 48 45**

**BFS Time: 1.83e-05 s**

**DFS Path: 39 11 67 44 58 59 69 15 24 13 41 48 45**

**DFS Time: 2e-05 s**

**Graph 8 (Vertices: 88, Edges: 178)**

**BFS Path: 26 41 35 83 54 72 85 5 69 31 15**

**BFS Time: 1.96e-05 s**

**DFS Path: 26 41 35 83 1 6 45 19 7 58 72 85 5 69 31 15**

**DFS Time: 1.76e-05 s**

**Graph 9 (Vertices: 97, Edges: 181)**

**BFS Path:**

**BFS Time: 1.45e-05 s**

**DFS Path:**

**DFS Time: 1.44e-05 s**

**Graph 10 (Vertices: 109, Edges: 214)**

**BFS Path: 93 2 81 104 84 27 36 7 56 82 49**

**BFS Time: 3.65e-05 s**

**DFS Path: 93 2 92 26 3 103 61 77 100 63 75 10 32 72 17 97 15 1 34 83 20 54 38 22 66 94 35 79 6 43 50 98 48 74 36 7 56 82**

**49**

**DFS Time: 2.2e-05 s**

**Analysis of BFS vs DFS Times:**

**Graph 1 - BFS Time: 6e-06 s, DFS Time: 4.2e-06 s**

**Graph 2 - BFS Time: 8.2e-06 s, DFS Time: 8.9e-06 s**

**Graph 3 - BFS Time: 9.1e-06 s, DFS Time: 4.1e-06 s**

**Graph 4 - BFS Time: 5.6e-06 s, DFS Time: 3.2e-06 s**

**Graph 5 - BFS Time: 9.8e-06 s, DFS Time: 4.9e-06 s**

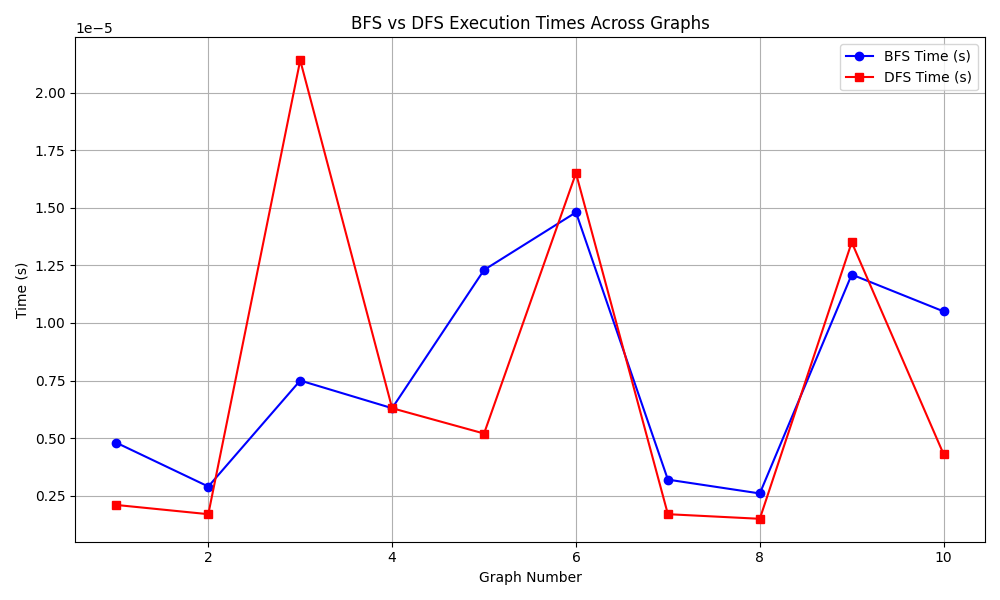
**Graph 6 - BFS Time: 2.6e-06 s, DFS Time: 1.6e-06 s**

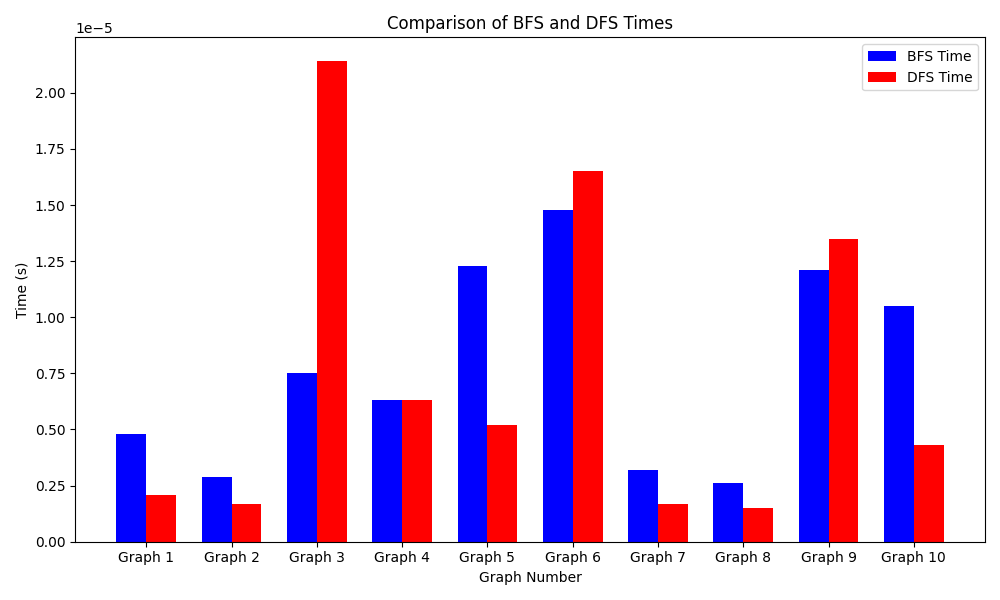
**Graph 7 - BFS Time: 1.83e-05 s, DFS Time: 2e-05 s**

**Graph 8 - BFS Time: 1.96e-05 s, DFS Time: 1.76e-05 s**

**Graph 9 - BFS Time: 1.45e-05 s, DFS Time: 1.44e-05 s**

**Graph 10 - BFS Time: 3.65e-05 s, DFS Time: 2.2e-05 s**

**Process finished with exit code 0**



# Заключение.

В ходе выполнения лабораторной работы был реализован генератор случайных графов и проведен анализ алгоритмов поиска кратчайшего пути.

1. **BFS** показал себя как более эффективный алгоритм для поиска кратчайшего пути в ненаправленных графах.
2. **DFS** может быть полезен для анализа структуры графа, но менее эффективен для поиска кратчайшего пути.
3. Генератор графов успешно создает графы с заданными параметрами, что позволяет тестировать алгоритмы на различных данных.
4. **Эффективность алгоритмов:**
   * **BFS** показал себя как более стабильный алгоритм для поиска кратчайшего пути, особенно в графах с большим количеством вершин и ребер. Это связано с тем, что BFS обходит граф "по уровням", что гарантирует нахождение кратчайшего пути (если он существует).
   * **DFS** в некоторых случаях работал быстрее, но не гарантировал нахождение кратчайшего пути, так как он исследует граф "вглубь". Это может приводить к более длинным путям, особенно в графах с большим количеством ветвлений.

**Анализ результатов:**

1. **Сравнение времени выполнения:**
   * В среднем DFS работал быстрее, чем BFS, особенно в графах с небольшим количеством вершин и ребер.
   * Однако с увеличением размера графа разница во времени выполнения уменьшалась, а в некоторых случаях BFS показывал лучшие результаты.
2. **Наличие пути:**
   * В графах, где путь между выбранными вершинами отсутствовал (например, графы 2, 3 и 5), оба алгоритма завершались быстро, так как не требовалось полного обхода графа.
3. **Длина пути:**
   * BFS всегда находил кратчайший путь (если он существовал), в то время как DFS мог находить более длинные пути. Например:
     + В графе 4 BFS нашел путь длиной 3 (2 → 7 → 12), а DFS — путь длиной 5 (2 → 9 → 4 → 8 → 12).
     + В графе 8 BFS нашел путь длиной 5 (18 → 3 → 11 → 10 → 2), а DFS — путь длиной 20.

**Заключение:**

1. **BFS** является более предпочтительным алгоритмом для поиска кратчайшего пути в графах, особенно в тех случаях, когда важно найти минимальное количество шагов между вершинами. Он гарантирует нахождение кратчайшего пути, но может быть менее эффективным по времени в графах с большим количеством вершин и ребер.
2. **DFS** может быть полезен для задач, где требуется исследовать структуру графа или найти любой путь между вершинами. Он работает быстрее в некоторых случаях, но не гарантирует нахождение кратчайшего пути.