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

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%204

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 19.03.2025

Оглавление

Описание задачи.

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, замерив время, требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

Описание графа.

Граф — это структура данных, состоящая из множества вершин (или узлов) и множества рёбер, которые соединяют эти вершины.

Основные компоненты графа:

- Вершины (или узлы) — объекты графа, которые могут представлять разные элементы системы (например, компьютеры в сети, города на карте и т. д.).
- Рёбра (или связи) — линии, которые соединяют вершины и могут иметь различные свойства, такие как направленность, вес или стоимость.

Основные понятия:

1. Степень вершины — количество рёбер, инцидентных вершине. В ненаправленном графе степень вершины — это число рёбер, соединяющих эту вершину с другими. В направленном графе различают:
    - Входящая степень — количество рёбер, направленных к вершине.
    - Исходящая степень — количество рёбер, исходящих от вершины.
2. Связность графа:
    - Связный граф — существует путь между любыми двумя вершинами графа.
    - Несвязный граф — существует хотя бы одна пара вершин, между которыми нет пути.
3. Путь — последовательность рёбер, соединяющих последовательность вершин.

Представления графа в памяти:

1. Матрица смежности:
    - Двумерный массив, где элемент в строке i и столбце j равен 1 (или весу ребра), если существует ребро между вершинами i и j, и 0 в противном случае.
    - Подходит для плотных графов, когда много рёбер.
2. Список смежности:
    - Это массив или список, где для каждой вершины хранится список её соседей.
    - Подходит для разреженных графов, когда количество рёбер значительно меньше количества возможных рёбер.
3. Список рёбер:
    - Это список всех рёбер графа, где каждое ребро представлено парой вершин, которые оно соединяет.
    - Подходит для хранения рёбер без необходимости в быстром поиске соседей.

Алгоритмы работы с графами:

1. Поиск в глубину (DFS) — исследует граф, начиная с вершины и углубляясь по пути, пока не встретит тупик.
2. Поиск в ширину (BFS) — исследует граф, посещая вершины на одном уровне (сначала все вершины, соседние с текущей, потом все вершины, соседние с ними)..

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace chrono;
```

- Подключаемые библиотеки
  - <iostream> — ввод и вывод результатов в консоль.
  - <vector> — динамический массив с доступом к элементам по индексу
  - <queue> — очередь для поисков.
  - <algorithm> — библиотека стандартных алгоритмов.
  - <chrono> — измерение времени.
- using namespace — упрощение кода, конфликта в данном проекте нет.

```
class RandomGraph {
private:
    int numVertices; // Количество вершин
    int numEdges;    // Количество рёбер
    bool directed;   // Направленность графа
    vector<vector<int>> adjMatrix;    // Матрица смежности
    vector<vector<int>> adjList;      // Список смежности
    vector<pair<int, int>> edges;     // Список рёбер

public:
    // Конструктор с параметрами генерации графа
    RandomGraph(int minVertices, int maxVertices, int minEdges, int maxEdges,
                int maxEdgesPerVertex, bool isDirected) {
        srand(time(0));

        numVertices = rand() % (maxVertices - minVertices + 1) + minVertices;
        numEdges = rand() % (maxEdges - minEdges + 1) + minEdges;
        directed = isDirected;

        // Инициализация структуры данных
        adjMatrix.resize(numVertices, vector<int>(numVertices, 0));
        adjList.resize(numVertices);

        generateGraph(maxEdgesPerVertex);
    }

    // Генерация графа
    void generateGraph(int maxEdgesPerVertex) {
```

```
        int edgeCount = 0;

        while (edgeCount < numEdges) {
            int u = rand() % numVertices;
            int v = rand() % numVertices;

            // Проверка на существование ребра (чтобы не дублировать рёбра)
            if (u != v && adjMatrix[u][v] == 0) {
                adjMatrix[u][v] = 1;
                adjList[u].push_back(v);
                edges.push_back({u, v});

                // Если граф направленный, то добавляем ребро только в одну
сторону
                if (!directed) {
                    adjMatrix[v][u] = 1;
                    adjList[v].push_back(u);
                    edges.push_back({v, u});
                }

                edgeCount++;
            }
        }
    }

    // Метод для поиска кратчайшего пути с помощью поиска в ширину (BFS)
    bool bfs(int start, int end) {
        vector<bool> visited(numVertices, false);
        vector<int> parent(numVertices, -1);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        while (!q.empty()) {
            int vertex = q.front();
            q.pop();

            for (int neighbor : adjList[vertex]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    parent[neighbor] = vertex;
                    q.push(neighbor);

                    if (neighbor == end) {
                        // Восстановление пути
                        vector<int> path;
                        for (int v = end; v != -1; v = parent[v]) {
                            path.push_back(v);
                        }
                        reverse(path.begin(), path.end());
                        cout << "Path (BFS): ";
                        for (int v : path) {
                            cout << v << " ";
```

```
                    }
                    cout << endl;
                    return true;
                }
            }
        }
    }

    return false;
}

// Метод для поиска кратчайшего пути с помощью поиска в глубину (DFS)
bool dfs(int start, int end, vector<bool>& visited, vector<int>& path) {
    visited[start] = true;
    path.push_back(start);

    if (start == end) {
        cout << "Path (DFS): ";
        for (int v : path) {
            cout << v << " ";
        }
        cout << endl;
        return true;
    }

    for (int neighbor : adjList[start]) {
        if (!visited[neighbor]) {
            if (dfs(neighbor, end, visited, path)) {
                return true;
            }
        }
    }

    path.pop_back();
    return false;
}

// Метод для вызова DFS
bool dfsWrapper(int start, int end) {
    vector<bool> visited(numVertices, false);
    vector<int> path;
    return dfs(start, end, visited, path);
}

// Методы для получения представлений графа
vector<vector<int>> getAdjMatrix() {
    return adjMatrix;
}

vector<vector<int>> getAdjList() {
    return adjList;
}

vector<pair<int, int>> getEdges() {
```

```
        return edges;
    }

    // Метод для отображения графа
    void displayGraph() {
        cout << "Adjacency Matrix: \n";
        for (auto& row : adjMatrix) {
            for (int val : row) {
                cout << val << " ";
            }
            cout << endl;
        }

        cout << "Adjacency List: \n";
        for (int i = 0; i < adjList.size(); ++i) {
            cout << i << ": ";
            for (int v : adjList[i]) {
                cout << v << " ";
            }
            cout << endl;
        }

        cout << "Edges: \n";
        for (auto& edge : edges) {
            cout << edge.first << " -> " << edge.second << endl;
        }
    }
};
```

- Класс RandomGraph — реализует генерацию и отображение случайного графа, а также поиска путей.
    - RandomGraph — конструктор, принимает параметры для генерации случайного графа, на выходе получаем граф, его матрицу и список смежности.
    - generateGraph — генерация ребер. Он проверяет, чтобы рёбра не дублировались, и если граф направленный, добавляет рёбра только в одну сторону (если граф не направленный — в обе).
    - Bfs — реализация алгоритма поиска в ширину для нахождения кратчайшего пути между двумя вершинами.
    - Dfs — реализация алгоритма поиска в глубину. Рекурсивный обход графа до тех пор, пока не будет найдена целевая вершина или не будут проверены все пути.
    - dfsWrapper — вспомогательный метод для удобства вызова поиска в глубину.
    - Методы для получения представлений графа:
    - displayGraph — getAdjMatrix(): возвращает матрицу смежности графа, getAdjList(): возвращает список смежности графа, getEdges(): Возвращает список рёбер графа.
    - displayGraph — вывод графа на экран в виде матрицы, списка смежности, и списка ребер.

```
void testGraphs(int minVertices, int maxVertices, int minEdges, int maxEdges,
                int maxEdgesPerVertex, bool directed) {
    for (int i = 0; i < 10; ++i) {
        // Количество вершин и рёбер для текущего графа
        int vertices = minVertices + i;
        int edges = minEdges + i;

        // Генерация графа
        RandomGraph graph(vertices, vertices + 1, edges, edges + 1,
maxEdgesPerVertex, directed);
        cout << "Generated graph with " << vertices << " vertices and " <<
edges << " edges.\n";
        graph.displayGraph();

        // Выбор случайных вершин для поиска пути
        int start = rand() % vertices;
        int end = rand() % vertices;
        cout << "Searching for path from " << start << " to " << end << "...\n";

        // Замер времени для поиска с использованием BFS
        auto start_time = high_resolution_clock::now();
        if (!graph.bfs(start, end)) {
            cout << "No path found using BFS." << endl;
        }
        auto end_time = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(end_time - start_time);
        cout << "BFS Time: " << duration.count() << " ns\n";

        // Замер времени для поиска с использованием DFS
        start_time = high_resolution_clock::now();
        if (!graph.dfsWrapper(start, end)) {
            cout << "No path found using DFS." << endl;
        }
        end_time = high_resolution_clock::now();
        duration = duration_cast<nanoseconds>(end_time - start_time);
        cout << "DFS Time: " << duration.count() << " ns\n";
    }
}
```

- Функция testGraphs генерирует необходимое кол-во графов, случайным образом выбирает вершины для поиска путей, а затем выполняет поиск в ширину и глубину, замеряет время в наносекундах и выводит на экран

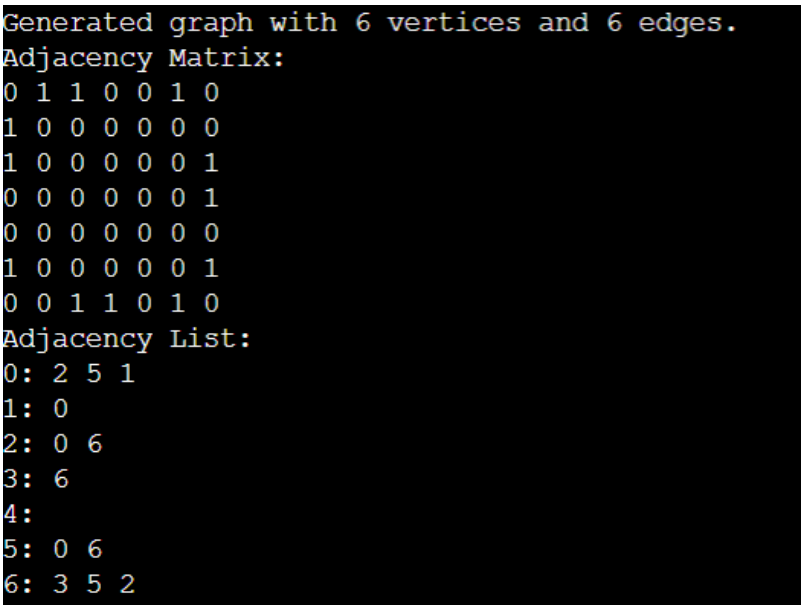
```
int main() {
    // Параметры генерации графов
    int minVertices = 5;
    int maxVertices = 15;
    int minEdges = 5;
    int maxEdges = 20;
    int maxEdgesPerVertex = 3;
    bool directed = false;
```

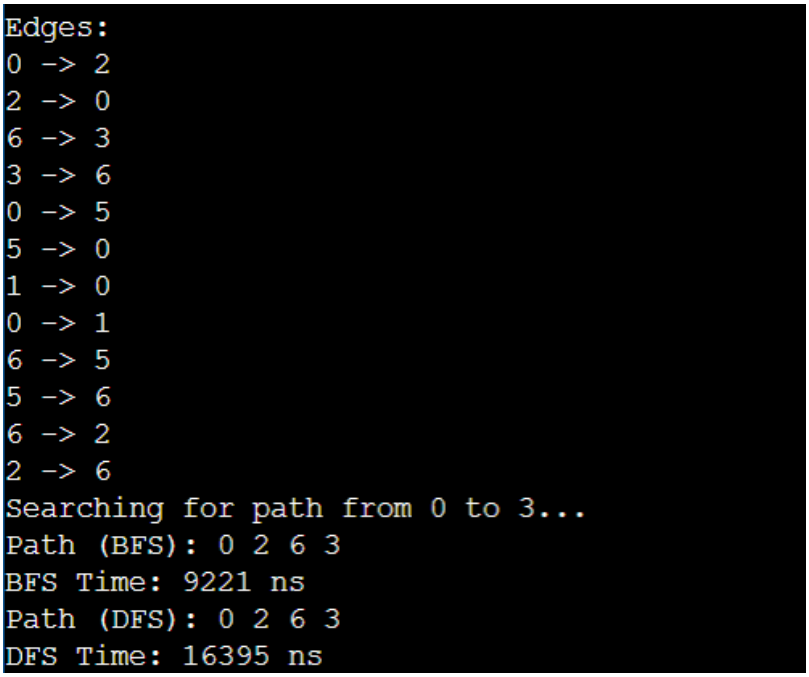
```
    testGraphs(minVertices, maxVertices, minEdges, maxEdges,
maxEdgesPerVertex, directed);

    return 0;
}
```

- Главная функция main, задаются параметры графа и вызывается функция тестирования графов.

Пример работы программы:



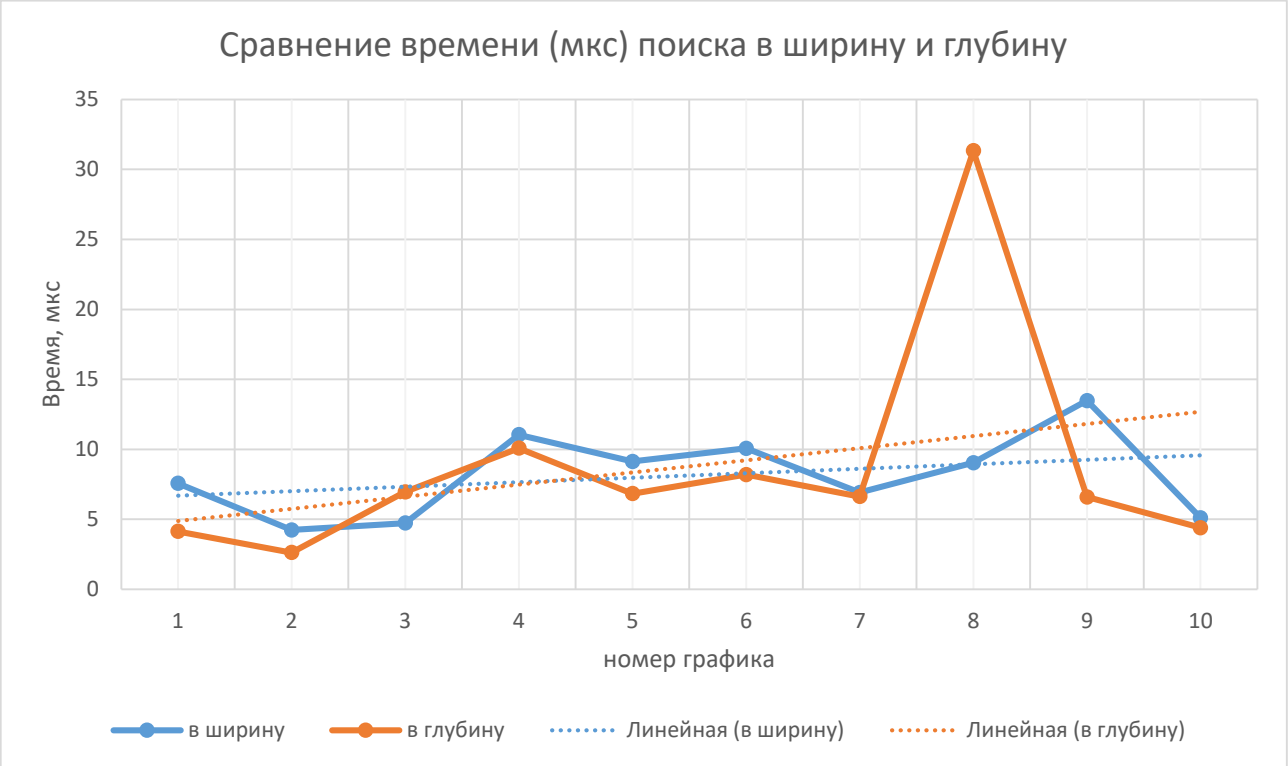


Заключение.

По результатам работы программы была собрана таблица:

			Время поиска (мкс)	
№	Вершины	Ребра	в ширину	в глубину
1	5	5	7,566	4,14
2	6	6	4,235	2,625
3	7	7	4,734	6,958
4	8	8	11,046	10,086
5	9	9	9,132	6,836
6	10	10	10,074	8,194
7	11	11	6,911	6,624
8	12	12	9,041	31,327
9	13	13	13,477	6,59
10	14	14	5,112	4,4

И по ней построен график. Для удобства последующего анализа были построены линии тренда:



Сравним два алгоритма:

1. Общие тенденции:
    - Поиск в ширину (BFS) и поиск в глубину (DFS) ведут себя по-разному в зависимости от размера графа.
    - В целом, поиск в глубину (DFS) показывает меньшее время на графах с маленьким числом вершин и рёбер, но с увеличением числа вершин и рёбер время резко увеличивается, особенно на более крупных графах.
    - Время работы поиска в ширину (BFS) относительно стабильно и в среднем немного больше, чем у DFS на малых графах, но не демонстрирует резких скачков.

2. Различия:
    - Для небольших графов (с 5-7 вершинами) поиск в глубину (DFS) оказывается более эффективным (меньшее время выполнения), чем поиск в ширину (BFS). Однако, это преимущество исчезает с увеличением размера графа.
    - Для более крупных графов (с 8 вершинами и более) поиск в ширину (BFS) начинает показывать более стабильное и предсказуемое время выполнения по сравнению с DFS. Резкие пики в времени работы DFS (например, на графе с 12 вершинами) говорят о том, что алгоритм сталкивается с высокими вычислительными затратами при глубоком обходе графа.
    - Особенности графов: Пиковые значения для некоторых графов (например, граф с 12 вершинами для DFS) могут быть связаны с их структурой. Возможно, для этих графов присутствуют циклы или сложные ветвления, что затрудняет работу DFS.

Таким образом, в зависимости от структуры графа, оба алгоритма имеют свои преимущества и недостатки. На малых графах DFS работает быстрее, но с увеличением числа вершин и рёбер BFS начинает показывать лучшие результаты.

Пример графа, построенного программой: