Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

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 вершиной.
- 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 списка смежности.
- Выдача списка ребер.

В качестве проверки работоспособности требуется сгенерировать 10 графов с возрастающим количеством вершин и рёбер (количество выбирать в зависимости от сложности расчёта 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 алгоритма

Под **графом** в математике понимается абстракция реальной системы объектов безотносительно их природы, обладающих парными связями. **Вершина графа** —

это некоторая точка, связанная с другими точками. **Ребро графа** — это линия, соединяющая две точки и олицетворяющая связь между ними.

Граф — это множество вершин, соединенных друг с другом произвольным образом множеством ребер

Что описывает граф:

- Взаимоотношение между людьми (социальные связи).
- Иерархические отношения (подчиненность людей, подразделений и прочего).
- Пути перемещения в любой местности (карта метро, сеть дорог).
- Взаимозависимости поставщиков услуг или товаров (поставщики для сборки одного автомобиля).
- Распределенные системы (любая микросервисная архитектура).
- Распределенные данные (реляционные базы данных).

Ориентированный граф — это граф, в котором каждое ребро имеет направление движения и, как правило, не предполагает возможности обратного перемещения. Направление ребер указывается стрелками.

Неориентированный граф — это граф, в котором ребра не имеют направления. Каждое ребро соединяет две вершины без указания порядка, и перемещение между ними возможно в обе стороны.

Путь (или маршрут) в графе — это последовательность вершин и ребер от начальной точки до конечной точки графа. Количество ребер, входящих в последовательность, задающую этот путь, называется **длиной пути**. **Циклический путь** — это путь, у которого начало совпадает с концом, и при этом его длина не равна нулю.

Для описания графа используют один из двух удобных для вычисления вариантов:

- Матрица смежности — это двумерная таблица, в которой строки и столбцы соответствуют вершинам, а значения в таблице соответствуют ребрам. Для

невзвешенного графа значения могут быть равны 1 (если связь есть и направлена в нужную сторону) или 0 (если связи нет). Для взвешенного графа в ячейках указываются конкретные значения весов.

- Матрица инцидентности — это матрица, в которой строки соответствуют вершинам, а столбцы — связям. В ячейках указывается 1, если связь выходит из вершины, -1, если связь входит в вершину, и 0 во всех остальных случаях.
- Список смежности — это список списков, содержащий все вершины. Внутренние списки для каждой вершины содержат все смежные с ней вершины.
- Список ребер — это список строк, в которых хранятся все ребра графа. Каждая строка содержит две вершины, соединенные этим ребром.

Обход графа — это переход от одной его вершины к другой с целью изучения свойств связей между вершинами. Выделяют два варианта обхода: **обход в глубину (DFS)** и **обход в ширину (BFS)**.

DFS (Depth-First Search, поиск в глубину) следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы движемся от начальной вершины в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или искомой вершины. Если конец пути не является искомой вершиной, мы возвращаемся к точке разветвления и продолжаем поиск по другому маршруту.

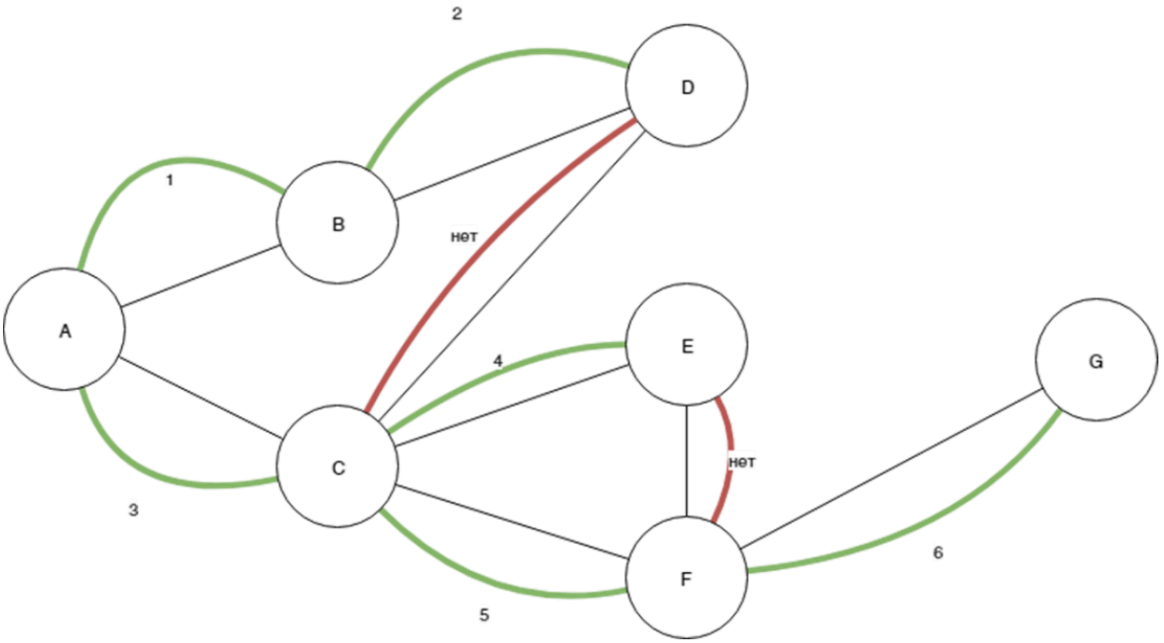

Рисунок 1 - Алгоритм поиска в глубину (DFS)

Алгоритм поиска в глубину (DFS):

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2. Запускаем процедуру dfs(u):
    a. Помечаем вершину u как пройденную.
    b. Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v).
3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Сложность алгоритма определяется количеством вершин и ребер в графе. Процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

В качестве дополнительной меры вершина может помечаться дважды:

- Первый раз, если она была посещена (окрашивается в серый цвет).
- Второй раз, если она полностью обработана (окрашивается в черный цвет).

BFS (Breadth-First Search, поиск в ширину) следует концепции «расширяйся, поднимаясь на высоту птичьего полета» («go wide, bird’s eye-view»). Вместо того

чтобы двигаться по одному пути до конца, BFS предполагает посещение ближайших соседей начальной вершины за один шаг, затем посещение соседей соседей и так далее, пока не будет найдена искомая вершина.

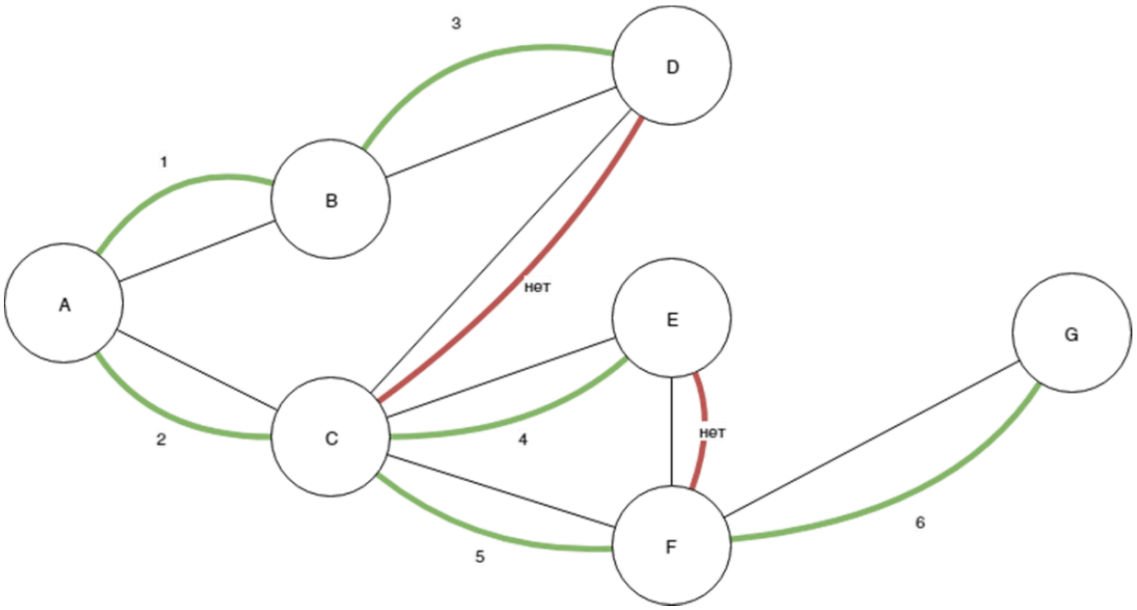

Рисунок 2 - Алгоритм поиска в ширину BFS

Алгоритм поиска в ширину (BFS):

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел u и пометить его как развернутый.
    a. Если узел u является целевым узлом, завершить поиск с результатом «успех».
    b. В противном случае в конец очереди добавляются все преемники узла u, которые еще не развернуты и не находятся в очереди.
3. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
4. Вернуться к шагу 2.

Сложность алгоритма также определяется количеством вершин и ребер в графе. Процедура вызывается для каждой вершины не более одного раза, а в рамках работы процедуры рассматриваются все ребра, исходящие из вершины.

Различия между DFS и BFS:

- Цель: поиск в ширину и в глубину — это способы обхода графа.
- Характер обхода:
    - DFS — это поиск по ребрам графа с возможностью возврата (туда-обратно).
    - BFS — это плавный обход по соседям, уровень за уровнем.
- Структура данных:
    - В DFS используется стек (явно или неявно через рекурсию).
    - В BFS используется очередь.
- Результат:
    - В DFS результатом является маршрут от стартовой вершины до искомой.
    - В BFS маршрут не всегда является результатом; алгоритм находит кратчайший путь в невзвешенных графах.
- Рекурсивность:
    - DFS является рекурсивным алгоритмом (или использует стек).
    - BFS не является рекурсивным и использует очередь.

Описание выполнения задачи

Для реализации задачи была написана программа на C++:

```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <iomanip>
#include <fstream>
#include <queue>
#include <stack>
#include <algorithm>

using namespace std;
```

```
// Класс Graph для представления графа
class Graph {
private:
    int vertices; // Количество вершин в графе
    vector<vector<int>> adj_matrix; // Матрица смежности: adj_matrix[i][j] = 1, если есть ребро из i в j
    vector<vector<int>> inc_matrix; // Матрица инцидентности: inc_matrix[i][j] показывает связь вершины i с ребром j
    vector<vector<int>> adj_list; // Список смежности: adj_list[i] содержит вершины, в которые ведут рёбра из i
    vector<pair<int, int>> edge_list; // Список рёбер: хранит пары (from, to) для каждого ребра
    bool is_directed; // Флаг, указывающий, является ли граф направленным

public:
    // Конструктор: инициализирует граф с заданным числом вершин и типом (направленный/ненаправленный)
    Graph(int v, bool dir) : vertices(v), is_directed(dir) {
        adj_matrix.resize(v, vector<int>(v, 0)); // Инициализация матрицы смежности нулями
        inc_matrix.resize(v, vector<int>(0)); // Инициализация матрицы инцидентности (пока пустая)
        adj_list.resize(v); // Инициализация списка смежности
    }

    // Проверка, существует ли ребро от вершины from к вершине to
    [[nodiscard]] bool hasEdge(int from, int to) const {
        return adj_matrix[from][to] == 1; // Возвращает true, если ребро существует
    }

    // Добавление ребра от вершины from к вершине to
    void addEdge(int from, int to) {
        // Отладочный вывод: показываем, какое ребро добавляем
        //cout << "Adding edge (" << from << ", " << to << ")\n";

        /*// Убеждаемся, что не добавляем петлю
        if (from == to) {
```

```
            cout << "Warning: Attempted to add a loop (" << from << ", " << to << "). Skipping.\n";
            return;
        }

        // Убеждаемся, что такого ребра ещё нет
        if (hasEdge(from, to)) {
            cout << "Warning: Edge (" << from << ", " << to << ") already exists. Skipping.\n";
            return;
        }*/

        // Добавляем ребро в матрицу смежности
        adj_matrix[from][to] = 1;
            if (!is_directed) adj_matrix[to][from] = 1; // Для ненаправленного графа добавляем обратное ребро

        // Добавляем ребро в список смежности
        adj_list[from].push_back(to);
        if (!is_directed) adj_list[to].push_back(from);

        // Добавляем ребро в список рёбер
        edge_list.emplace_back(from, to);

        // Обновляем матрицу инцидентности: добавляем новый столбец для нового ребра
        for (int i = 0; i < vertices; ++i) {
            if (i == from) inc_matrix[i].push_back(is_directed ? -1 : 1); // Из вершины from ребро выходит (-1 для направленного)
            else if (i == to) inc_matrix[i].push_back(1); // В вершину to ребро входит (1)
            else inc_matrix[i].push_back(0); // Остальные вершины не связаны с этим ребром
        }
    }

    // Вывод матрицы смежности
    void printAdjacencyMatrix() {
        cout << "Adjacency Matrix:\n";
        for (int i = 0; i < vertices; i++) {
```

```
        for (int j = 0; j < vertices; j++) {
            cout << adj_matrix[i][j] << " ";
        }
        cout << "\n";
    }
}

// Вывод матрицы инцидентности
void printIncidenceMatrix() {
    cout << "Incidence Matrix:\n";
    int num_edges = inc_matrix[0].size(); // Число рёбер — это число столбцов в матрице
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < num_edges; j++) {
            cout << setw(3) << inc_matrix[i][j]; // Выводим значение с выравниванием (ширина
3)
        }
        cout << "\n";
    }
}

// Вывод списка смежности
void printAdjacencyList() {
    cout << "Adjacency List:\n";
    for (int i = 0; i < vertices; i++) {
        cout << i << ": ";
        for (int j : adj_list[i]) {
            cout << j << " ";
        }
        cout << "\n";
    }
}

// Вывод списка рёбер
void printEdgeList() {
    cout << "Edge List:\n";
```

```
            for (auto& edge : edge_list) {
                cout << "(" << edge.first << ", " << edge.second << ") ";
            }
            cout << "\n";
        }

        // Получение количества вершин
        [[nodiscard]] int getVertices() const {
            return vertices;
        }

        // Получение списка смежности для заданной вершины
        vector<int>& getAdjList(int v) {
            return adj_list[v];
        }

        // Получение количества рёбер
        [[nodiscard]] int getEdgeCount() const {
            return edge_list.size(); // Размер списка рёбер — это число рёбер в графе
        }
    };

    // Функция для генерации случайного графа
    Graph generateRandomGraph(int v, int e, int max_edges_per_vertex, bool is_directed, int max_in_edges, int max_out_edges) {
        // Инициализация генератора случайных чисел
        random_device rd; // Источник случайности
        mt19937 gen(rd()); // Генератор случайных чисел (Mersenne Twister)

        // Создаём граф с заданным числом вершин и типом (направленный/ненаправленный)
        Graph graph(v, is_directed);

        // Если вершина одна или меньше, добавлять рёбра невозможно, возвращаем пустой граф
        if (v <= 1) return graph;
```

```
// Массивы для отслеживания степеней вершин
   vector<int> out_degrees(v, 0); // Исходящая степень каждой вершины (сколько рёбер выходит из вершины)
   vector<int> in_degrees(v, 0); // Входящая степень каждой вершины (сколько рёбер входит в вершину)
   vector<int> total_degrees(v, 0); // Общая степень каждой вершины (входящие + исходящие рёбра)

// Создаём равномерное распределение для выбора случайных вершин от 0 до v-1
uniform_int_distribution<> vertex_choice(0, v - 1);

// Переменные для контроля процесса добавления рёбер
int attempts = 0; // Счётчик попыток добавления рёбер
 int max_attempts = 10000; // Максимальное число попыток, чтобы избежать бесконечного цикла
int added_edges = 0; // Счётчик успешно добавленных рёбер

 // Основной цикл: пытаемся добавить рёбра, пока не достигнем целевого числа e или не превысим max_attempts
while (added_edges < e && attempts < max_attempts) {
   // Случайно выбираем начальную вершину (from) и конечную вершину (to)
   int from = vertex_choice(gen);
   int to = vertex_choice(gen);
   attempts++; // Увеличиваем счётчик попыток

      // Проверяем, можно ли добавить ребро (from, to). Условия, при которых ребро добавить нельзя:
   // 1. from и to — одна и та же вершина (петли запрещены)
   // 2. Ребро (from, to) уже существует
   // 3. Общее число рёбер, связанных с вершиной from, достигло max_edges_per_vertex
   // 4. Общее число рёбер, связанных с вершиной to, достигло max_edges_per_vertex
   // 5. Для направленного графа: исходящая степень from превышает max_out_edges
   // 6. Для направленного графа: входящая степень to превышает max_in_edges
   if (from == to || graph.hasEdge(from, to) ||
      total_degrees[from] >= max_edges_per_vertex ||
```

```
                total_degrees[to] >= max_edges_per_vertex ||
                        (is_directed && (out_degrees[from] >= max_out_edges || in_degrees[to] >=
max_in_edges))) {
                    continue; // Пропускаем попытку, если хотя бы одно из условий выполнено
                }

                // Если все проверки пройдены, добавляем ребро
                graph.addEdge(from, to);

                // Обновляем степени вершин
                out_degrees[from]++; // Увеличиваем исходящую степень вершины from
                total_degrees[from]++; // Увеличиваем общее число рёбер, связанных с from
                if (is_directed) {
                    in_degrees[to]++; // Для направленного графа увеличиваем входящую степень to
                    total_degrees[to]++; // Увеличиваем общее число рёбер, связанных с to
                } else {
                    out_degrees[to]++; // Для ненаправленного графа увеличиваем исходящую степень to
                    total_degrees[to]++; // Увеличиваем общее число рёбер, связанных с to
                }
                added_edges++; // Увеличиваем счётчик добавленных рёбер
            }

            // Если не удалось добавить все рёбра за max_attempts попыток, граф всё равно
возвращается
            return graph; // Возвращаем сгенерированный граф
        }

        // Поиск в ширину (BFS) для нахождения кратчайшего пути между start и end
        int bfs(Graph& g, int start, int end, vector<int>& path) {
            int v_count = g.getVertices(); // Получаем количество вершин
            vector<int> distance(v_count, -1); // Массив расстояний: -1 означает, что вершина не
посещена
            vector<int> parent(v_count, -1); // Массив родителей для восстановления пути
            queue<int> q; // Очередь для BFS
```

```
    distance[start] = 0; // Расстояние до начальной вершины равно 0
    q.push(start); // Добавляем начальную вершину в очередь

    // Основной цикл BFS
    while (!q.empty()) {
        int u = q.front(); // Извлекаем вершину из начала очереди
        q.pop();
        for (int v : g.getAdjList(u)) { // Перебираем всех соседей вершины u
            if (distance[v] == -1) { // Если вершина v ещё не посещена
                distance[v] = distance[u] + 1; // Устанавливаем расстояние до v
                parent[v] = u; // Сохраняем родителя v для восстановления пути
                q.push(v); // Добавляем v в очередь
            }
        }
    }

    // Если путь до конечной вершины не найден, возвращаем -1
    if (distance[end] == -1) return -1;

    // Восстанавливаем путь от end до start
    int curr = end;
    while (curr != -1) {
        path.push_back(curr); // Добавляем текущую вершину в путь
        curr = parent[curr]; // Переходим к родителю
    }
    if (!path.empty()) { // Если путь не пустой, разворачиваем его (чтобы он шёл от start к end)
        reverse(path.begin(), path.end());
    }
    return distance[end]; // Возвращаем длину кратчайшего пути
}

// Поиск в глубину (DFS) для нахождения пути между start и end
int dfs(Graph& g, int start, int end, vector<int>& path) {
    int v_count = g.getVertices(); // Получаем количество вершин
    vector<bool> visited(v_count, false); // Массив посещённых вершин
```

```
vector<int> parent(v_count, -1); // Массив родителей для восстановления пути
stack<int> s; // Стек для итеративного DFS
s.push(start); // Начинаем с вершины start

// Основной цикл DFS
while (!s.empty()) {
    int u = s.top(); // Извлекаем вершину из вершины стека
    s.pop();

    if (u == end) { // Если достигли конечной вершины, завершаем поиск
        visited[u] = true;
        break;
    }

    if (!visited[u]) { // Если вершина u ещё не посещена
        visited[u] = true; // Помечаем её как посещённую
        // Получаем соседей вершины u
        vector<int> neighbors = g.getAdjList(u);
        // Разворачиваем соседей, чтобы порядок обхода был более предсказуемым
        reverse(neighbors.begin(), neighbors.end());
        for (int v : neighbors) { // Перебираем соседей
            if (!visited[v]) { // Если сосед v не посещён
                parent[v] = u; // Сохраняем родителя v для восстановления пути
                s.push(v); // Добавляем v в стек
            }
        }
    }
}

// Если конечная вершина не посещена, путь не существует
if (!visited[end]) return -1;

// Восстанавливаем путь от end до start
int curr = end;
while (curr != -1) {
```

```
            path.push_back(curr); // Добавляем текущую вершину в путь
            curr = parent[curr]; // Переходим к родителю
        }
        if (!path.empty()) { // Если путь не пустой, разворачиваем его
            reverse(path.begin(), path.end());
        }
        return path.size() - 1; // Возвращаем длину пути
    }

    // Функция для записи данных о графах в CSV-файл
    void generateGraphData(const vector<double>& bfs_times, const vector<double>& dfs_times, const vector<int>& sizes, const vector<bool>& directed, const vector<int>& edges) {
        ofstream file("graph_data.csv"); // Открываем файл для записи
        if (file.is_open()) {
            // Записываем заголовок CSV
            file << "Graph,Vertices,Edges,Directed,BFS_Time,DFS_Time\n";
            // Записываем данные для каждого графа
            for (size_t i = 0; i < sizes.size(); ++i) {
                file << i + 1 << "," << sizes[i] << "," << edges[i] << "," << (directed[i] ? "Yes" : "No") << "," << bfs_times[i] << "," << dfs_times[i] << "\n";
            }
            file.close();
            cout << "Graph data saved to graph_data.csv. Use a plotting tool (e.g., Python) to visualize.\n";
        }
    }

    int main() {
        random_device rd; // Генератор случайных чисел
        mt19937 gen(rd());

        // Параметры для генерации графов
        int min_vertices = 5, max_vertices = 10; // Диапазон числа вершин
        int min_edges = 10, max_edges = 20; // Диапазон числа рёбер
        int step_vertices = (max_vertices - min_vertices) / 9; // Шаг увеличения числа вершин
```

```
int step_edges = (max_edges - min_edges) / 9; // Шаг увеличения числа рёбер
  int max_edges_per_vertex = 5; // Максимальная общая степень вершины (входящие + исходящие рёбра)
  int max_in_edges = 3; // Максимальное количество входящих рёбер для направленного графа
  int max_out_edges = 3; // Максимальное количество исходящих рёбер для направленного графа

// Векторы для хранения данных о графах
vector<int> sizes; // Размеры графов (число вершин)
vector<bool> directed_flags; // Флаги направленности графов
vector<Graph> graphs; // Сами графы
vector<double> bfs_times, dfs_times; // Времена выполнения BFS и DFS
vector<int> edge_counts; // Количество рёбер в каждом графе

// Генерация 10 направленных графов
cout << "Generating Directed Graphs:\n";
for (int i = 0; i < 10; ++i) {
   // Вычисляем число вершин и рёбер для текущего графа
   int v = min_vertices + i * step_vertices;
   int e = min_edges + i * step_edges;
   e = min(e, v * (v - 1)); // Ограничиваем число рёбер для направленного графа
   sizes.push_back(v); // Сохраняем размер графа
   directed_flags.push_back(true); // Указываем, что граф направленный

   // Генерируем граф
      Graph g = generateRandomGraph(v, e, max_edges_per_vertex, true, max_in_edges, max_out_edges);
   graphs.push_back(g); // Сохраняем граф
   edge_counts.push_back(g.getEdgeCount()); // Сохраняем реальное количество рёбер

   // Выбираем случайные начальную и конечную вершины для поиска пути
   uniform_int_distribution<> vertex_choice(0, v - 1);
   int start = vertex_choice(gen);
   int end = vertex_choice(gen);
```

```
            while (start == end) end = vertex_choice(gen); // Гарантируем, что start != end

            // Выполняем BFS
            vector<int> bfs_path;
            auto bfs_start = chrono::high_resolution_clock::now(); // Засекаем время начала
            int bfs_dist = bfs(g, start, end, bfs_path); // Ищем кратчайший путь
            auto bfs_end = chrono::high_resolution_clock::now(); // Засекаем время окончания
                double bfs_time = chrono::duration<double>(bfs_end - bfs_start).count(); // Вычисляем
время выполнения
            bfs_times.push_back(bfs_time); // Сохраняем время выполнения BFS
            cout << "Directed Graph " << i + 1 << " (V=" << v << ", E=" << g.getEdgeCount() << "):\n";
            cout << "BFS from " << start << " to " << end << ": ";
            if (bfs_dist == -1) cout << "No path\n"; // Если пути нет
            else {
               cout << "Shortest Distance = " << bfs_dist << ", Path = "; // Выводим длину пути
               for (int vertex : bfs_path) cout << vertex << " "; // Выводим сам путь
            }
            cout << "\nTime: " << bfs_time << "s\n";

            // Выполняем DFS
            vector<int> dfs_path;
            auto dfs_start = chrono::high_resolution_clock::now();
            int dfs_dist = dfs(g, start, end, dfs_path);
            auto dfs_end = chrono::high_resolution_clock::now();
            double dfs_time = chrono::duration<double>(dfs_end - dfs_start).count();
            dfs_times.push_back(dfs_time);
            cout << "DFS from " << start << " to " << end << ": ";
            if (dfs_dist == -1) cout << "No path\n";
            else {
               cout << "Path Length = " << dfs_dist << ", Path = ";
               for (int vertex : dfs_path) cout << vertex << " ";
            }
            cout << "\nTime: " << dfs_time << "s\n\n";

            // Для первого графа выводим дополнительные данные
```

```
            if (i == 0) {
                g.printAdjacencyMatrix();
                g.printIncidenceMatrix();
                g.printAdjacencyList();
                g.printEdgeList();
            }
        }

        // Генерация 10 ненаправленных графов
        cout << "Generating Undirected Graphs:\n";
        for (int i = 0; i < 10; ++i) {
            int v = min_vertices + i * step_vertices;
            int e = min_edges + i * step_edges;
            e = min(e, v * (v - 1) / 2); // Ограничиваем число рёбер для ненаправленного графа
            sizes.push_back(v);
            directed_flags.push_back(false);

            Graph g = generateRandomGraph(v, e, max_edges_per_vertex, false, max_in_edges, max_out_edges);
            graphs.push_back(g);
            edge_counts.push_back(g.getEdgeCount());

            uniform_int_distribution<> vertex_choice(0, v - 1);
            int start = vertex_choice(gen);
            int end = vertex_choice(gen);
            while (start == end) end = vertex_choice(gen);

            vector<int> bfs_path;
            auto bfs_start = chrono::high_resolution_clock::now();
            int bfs_dist = bfs(g, start, end, bfs_path);
            auto bfs_end = chrono::high_resolution_clock::now();
            double bfs_time = chrono::duration<double>(bfs_end - bfs_start).count();
            bfs_times.push_back(bfs_time);
             cout << "Undirected Graph " << i + 1 << " (V=" << v << ", E=" << g.getEdgeCount() << "):\n";
```

```
    cout << "BFS from " << start << " to " << end << ": ";
    if (bfs_dist == -1) cout << "No path\n";
    else {
        cout << "Shortest Distance = " << bfs_dist << ", Path = ";
        for (int vertex : bfs_path) cout << vertex << " ";
    }
    cout << "\nTime: " << bfs_time << "s\n";

    vector<int> dfs_path;
    auto dfs_start = chrono::high_resolution_clock::now();
    int dfs_dist = dfs(g, start, end, dfs_path);
    auto dfs_end = chrono::high_resolution_clock::now();
    double dfs_time = chrono::duration<double>(dfs_end - dfs_start).count();
    dfs_times.push_back(dfs_time);
    cout << "DFS from " << start << " to " << end << ": ";
    if (dfs_dist == -1) cout << "No path\n";
    else {
        cout << "Path Length = " << dfs_dist << ", Path = ";
        for (int vertex : dfs_path) cout << vertex << " ";
    }
    cout << "\nTime: " << dfs_time << "s\n\n";

    if (i == 0) {
        g.printAdjacencyMatrix();
        g.printIncidenceMatrix();
        g.printAdjacencyList();
        g.printEdgeList();
    }
}

generateGraphData(bfs_times, dfs_times, sizes, directed_flags, edge_counts);

double avg_bfs_directed = 0, avg_dfs_directed = 0;
double avg_bfs_undirected = 0, avg_dfs_undirected = 0;
for (size_t i = 0; i < bfs_times.size(); ++i) {
```

```
            if (directed_flags[i]) {
                avg_bfs_directed += bfs_times[i];
                avg_dfs_directed += dfs_times[i];
            } else {
                avg_bfs_undirected += bfs_times[i];
                avg_dfs_undirected += dfs_times[i];
            }
        }
        avg_bfs_directed /= 10;
        avg_dfs_directed /= 10;
        avg_bfs_undirected /= 10;
        avg_dfs_undirected /= 10;

        cout << "Analysis:\n";
        cout << "Directed Graphs:\n";
        cout << "Average BFS time: " << avg_bfs_directed << "s\n";
        cout << "Average DFS time: " << avg_dfs_directed << "s\n";
        cout << "Undirected Graphs:\n";
        cout << "Average BFS time: " << avg_bfs_undirected << "s\n";
        cout << "Average DFS time: " << avg_dfs_undirected << "s\n";
         cout << "BFS finds the shortest path efficiently, while DFS may fail if vertices are in different components.\n";

        return 0;
    }
```

Данный код представляет собой реализацию программы на языке C++, которая моделирует и анализирует графы различных типов (направленные и ненаправленные) с использованием различных структур данных и алгоритмов поиска. Основная цель программы — генерация случайных графов с заданными параметрами (число вершин и ребер), выполнение алгоритмов поиска в ширину (BFS) и в глубину (DFS) для нахождения путей между случайными парами вершин, а также запись результатов в файл формата CSV для последующего анализа. Программа также предоставляет возможность визуализации структуры графа через

вывод матрицы смежности, матрицы инцидентности, списка смежности и списка ребер.

Код организован следующим образом:

- Класс Graph: Определяет граф с использованием четырёх структур данных: матрицы смежности (adj_matrix), матрицы инцидентности (inc_matrix), списка смежности (adj_list) и списка рёбер (edge_list). Класс поддерживает как направленные, так и ненаправленные графы.
- Функции генерации и поиска:
    - generateRandomGraph: Генерирует случайный граф с заданным числом вершин и рёбер, учитывая ограничения на степени вершин.
    - bfs: Реализует алгоритм поиска в ширину для нахождения кратчайшего пути.
    - dfs: Реализует итеративный алгоритм поиска в глубину для нахождения любого пути.
    - generateGraphData: Записывает результаты в файл graph_data.csv.
- Основной цикл (main): Организует генерацию 10 направленных и 10 ненаправленных графов, выполняет измерение времени выполнения BFS и DFS, выводит результаты и сохраняет данные.

Специфические элементы реализации:

- Многоструктурное представление графа.
    - Граф представлен одновременно четырьмя структурами: матрица смежности используется для проверки наличия ребер, матрица инцидентности — для анализа связей вершин с ребрами, список смежности — для эффективного обхода соседей, а список ребер — для подсчёта общего числа ребер. Это позволяет гибко использовать граф в различных задачах, хотя добавление каждой структуры увеличивает потребление памяти.

- При добавлении ребра (addEdge) все структуры обновляются синхронно, что обеспечивает согласованность данных.

- Генерация случайного графа.
  - Функция generateRandomGraph использует генератор случайных чисел (mt19937) для выбора вершин и пытается добавить ребра до достижения целевого числа или исчерпания максимального числа попыток (max_attempts = 10000). Ограничения на максимальную степень вершин (max_edges_per_vertex) и входящую/исходящую степень (max_in_out_edges) предотвращают перегрузку отдельных вершин, но могут приводить к тому, что реальное число рёбер будет меньше целевого.
  - Для направленных графов максимальное число рёбер ограничено $v \times (v - 1)$, а для ненаправленных — $v \times (v - 1) / 2$, что соответствует теоретическому максимуму ребер.
- Итеративная реализация DFS.
  - Алгоритм DFS реализован итеративно с использованием стека (std::stack) вместо рекурсии. Это улучшает производительность на больших графах, избегая переполнения стека вызовов. Соседи вершины добавляются в стек в обратном порядке (reverse), что обеспечивает предсказуемый порядок обхода.
  - Восстановление пути осуществляется через массив родителей (parent), что позволяет возвращать конкретный путь, а не только факт его существования.
- Измерение производительности.
  - Время выполнения BFS и DFS измеряется с использованием chrono::high_resolution_clock, что обеспечивает высокую точность (в секундах с высокой разрядностью).
  - Средние значения времени вычисляются отдельно для направленных и ненаправленных графов, что позволяет сравнить эффективность алгоритмов в зависимости от типа графа.

- Вывод и сохранение данных.
    - Файл graph_data.csv содержит столбцы: Graph (номер графа), Size (число вершин), Edges (реальное число рёбер), Directed (направленность), BFS_Time и DFS_Time. Это позволяет использовать данные для визуализации (например, с помощью Python и matplotlib).
    - Вывод в консоль включает подробную информацию о каждом графе, включая путь и время выполнения, а для первого графа каждого типа также отображаются все представления структуры.
- Оптимизация и ограничения:
    - Код включает проверку на пустоту пути (if (!path.empty())) перед разворотом, что предотвращает ошибки при отсутствии пути.
    - Для больших графов (например, *v* от 1000 до 10000, *e* от 5000 до 50000) может наблюдаться замедление из-за роста памяти и числа операций. Ограничение на попытки добавления ребер помогает избежать бесконечных циклов.

Программа демонстрирует реализацию базовых алгоритмов работы с графами, предоставляя гибкие инструменты для их анализа. Специфические элементы, такие как итеративный DFS и многоструктурное представление, делают код полезным для образовательных целей и экспериментов с графами различной сложности. Данные, сохраненные в CSV, позволяют провести дальнейший анализ производительности и структуры графов с использованием внешних инструментов.

Выводы

В ходе выполнения лабораторной работы был реализована генерация направленных и ненаправленных графов, возможность вывода матриц инцидентности и смежности, а также списки смежности и вершин. На основании генерации 10 направленных и 10 ненаправленных графов были проанализированы BFS и DFS, были выяснены кратчайшие пути для случайно выбранных вершин.

Graph	Vertices	Edges	Directed	BFS_Time	DFS_Time
1	1000	2759	Yes	0,0001104	0,0003650

2	2000	4894	Yes	0,0002332	0,0002516
3	3000	6472	Yes	0,0002453	0,0084631
4	4000	7500	Yes	0,0003649	0,0008658
5	5000	8271	Yes	0,0000376	0,0000213
6	6000	8750	Yes	0,0000386	0,0000230
7	7000	9067	Yes	0,0007251	0,0027708
8	8000	9277	Yes	0,0002476	0,0005854
9	9000	9476	Yes	0,0000461	0,0000613
10	10000	9576	Yes	0,0000956	0,0001215
11	1000	2945	No	0,0003466	0,0005313
12	2000	5534	No	0,0002356	0,0002733
13	3000	7343	No	0,0004417	0,0012444
14	4000	8374	No	0,0004419	0,0016573
15	5000	9067	No	0,0005032	0,0005029
16	6000	9409	No	0,0005647	0,0007408
17	7000	9623	No	0,0006658	0,0014210
18	8000	9735	No	0,0000762	0,0000901
19	9000	9825	No	0,0007418	0,0038622
20	10000	9887	No	0,0011121	0,0054135

Таблица 1 - Полученные в результате работы программы данные о времени выполнения BFS и DFS для направленных и ненаправленных графов

Были построены графики зависимости времени выполнения алгоритмов BFS и DFS от вершин для направленных (рис. 3) и ненаправленных (рис. 4) графов при помощи полученных в результате работы программы данных, представленных в таблице 1.

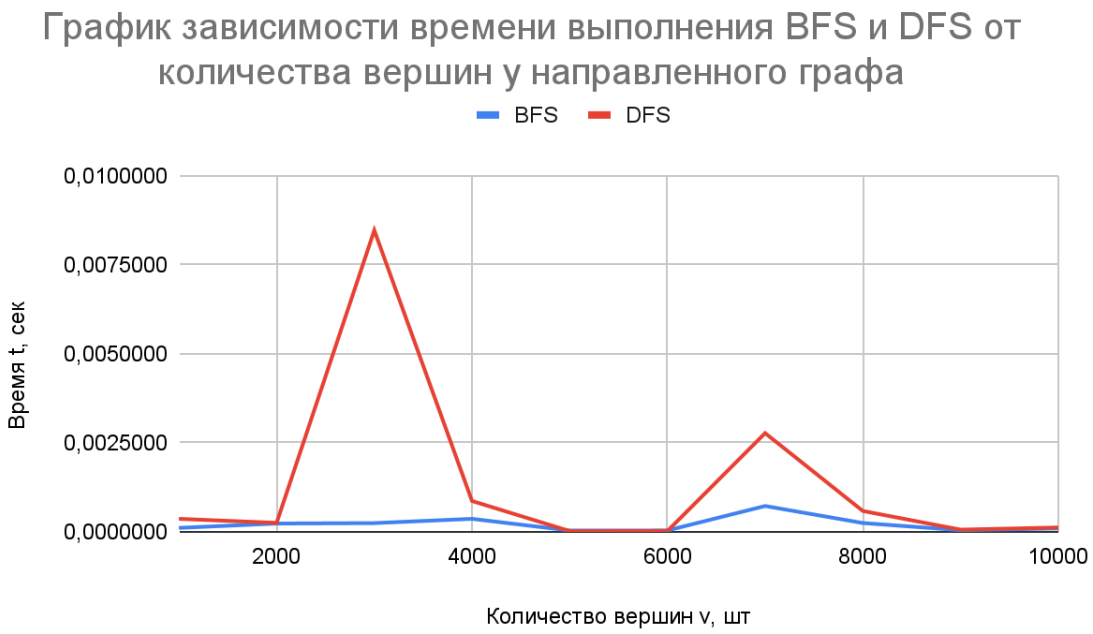

Рисунок 3 - График зависимости времени выполнения алгоритмов BFS и DFS от вершин для направленных графов

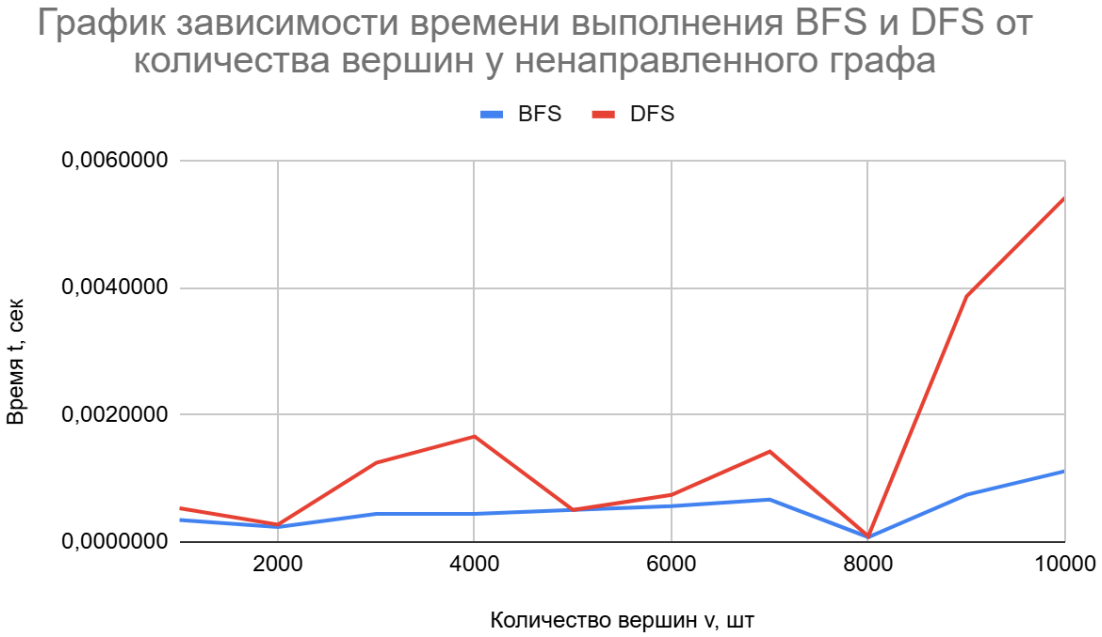

Рисунок 4 - График зависимости времени выполнения алгоритмов BFS и DFS от вершин для ненаправленных графов

Также программа подсчитала среднее время выполнения алгоритмов BFS и DFS:

- Для направленных графов:
  - BFS: 0,00021444 секунд;
  - DFS: 0,00135288 секунд;

- Для ненаправленных графов:
    - BFS: 0,00051296 секунд;
    - DFS: 0,00157368 секунд;

Анализ результатов показывает, что BFS в среднем работает быстрее, чем DFS, как на направленных (0.00021444 секунд против 0.00135288 секунд), так и на ненаправленных графах (0.00051296 секунд против 0.00157368 секунд). Это объясняется тем, что BFS ориентирован на поиск кратчайшего пути и исследует граф послойно, что позволяет быстрее находить путь, особенно в разреженных графах. DFS, напротив, углубляется в одну ветвь графа, что может приводить к более длительному поиску, особенно если путь находится в другой компоненте связности.

Также на основании графика зависимости времени выполнения BFS и DFS от числа вершин можно сделать вывод, что время выполнения BFS растет более плавно, чем DFS, особенно для больших графов. DFS показывает большую вариативность времени выполнения, что связано с его зависимостью от структуры графа и порядка обхода вершин. На ненаправленных графах оба алгоритма работают медленнее, чем на направленных, что может быть связано с большим числом рёбер и более плотной структурой.

В ходе лабораторной работы был успешно реализован генератор случайных графов с заданными параметрами, а также класс Graph с необходимыми методами представления графа. Проведенные эксперименты показали, что BFS является эффективным алгоритмом для поиска кратчайшего пути в графах с заданными характеристиками, особенно на направленных графах. DFS, несмотря на большую вариативность времени выполнения, может быть полезен в задачах, где требуется просто найти любой путь, а не обязательно кратчайший.