Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

По дисциплине «Объектно-ориентированное программирование»

Обход графа в глубину, в ширину, Топологическая сортировка (Kahn’s algorithm)

(семестр VI)

Студент группы
3331506/20401 М. А. Аксенов

подпись, дата инициалы и фамилия

Оценка выполненной студентом работы:

Преподаватель,
доцент, к.т.н. М. С. Ананьевский

подпись, дата инициалы и фамилия

Санкт-Петербург
2025

СОДЕРЖАНИЕ

Введение

Поиск в глубину (англ. Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину.

Поиск в ширину (англ. *Breadth-first search*, BFS) — один из методов обхода графа. Пусть задан граф и выделена исходная вершина s. Алгоритм поиска в ширину исследует граф слоями, начиная от начальной вершины.

Поиск в ширину имеет такое название потому, что в процессе обхода мы идём вширь, то есть перед тем как приступить к поиску вершин на расстоянии $k + 1$, выполняется обход вершин на расстоянии k.

Топологическая сортировка (Kahn's algorithm) упорядочивает вершины ориентированного ациклического графа (DAG) так, чтобы каждое ребро вело от вершины с меньшим порядковым номером к вершине с большим.

В рамках курсовой работы будут разработаны 3 программы на языке C++. А также будет проведён анализ скорости выполнения алгоритмов. При этом стоит учесть, что конечный код должен быть читаем и понятен, обработку ошибок необходимо реализовать через исключения, а какие-либо выводы в консоль, кроме режима отладки, запрещены.

Актуальность работы обусловлена необходимостью эффективной обработки графов в современных приложениях, таких как социальные сети, системы рекомендаций и логистика.

Основная часть

1. Обход в глубину (DFS)

На примере простого графа разберём принцип работы DFS более подробно. Пусть дан следующий граф. Начнём обход с узла A, и он сразу будет пройденным.

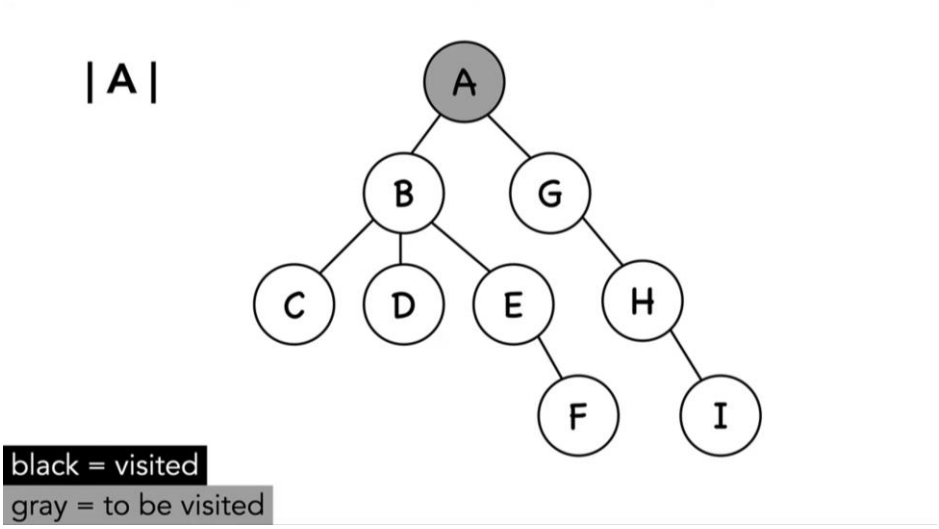

Рисунок 1 – Начальный узел

Добавляем соседние узлы B и G в стек. Пойдём, например, в левую ветвь, т. е. B – следующий узел, извлечённый из стека, пометим его как пройденный.

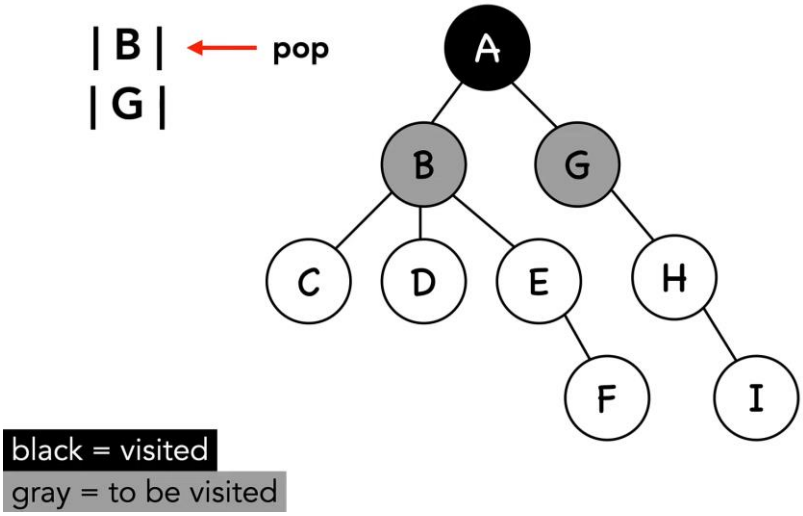

Рисунок 2 – Добавление соседей в стек

Узел G остаётся на дне стека.

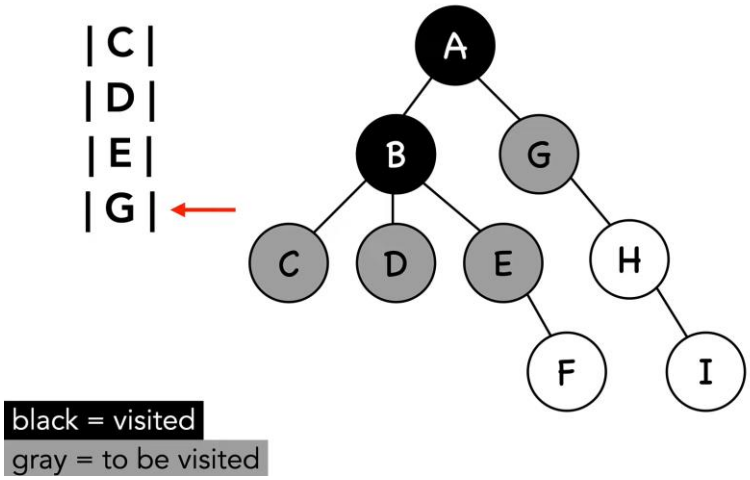

Рисунок 3 – Попали в узел B

Далее идём в верхний элемент стека – точку C и убираем её из стека.

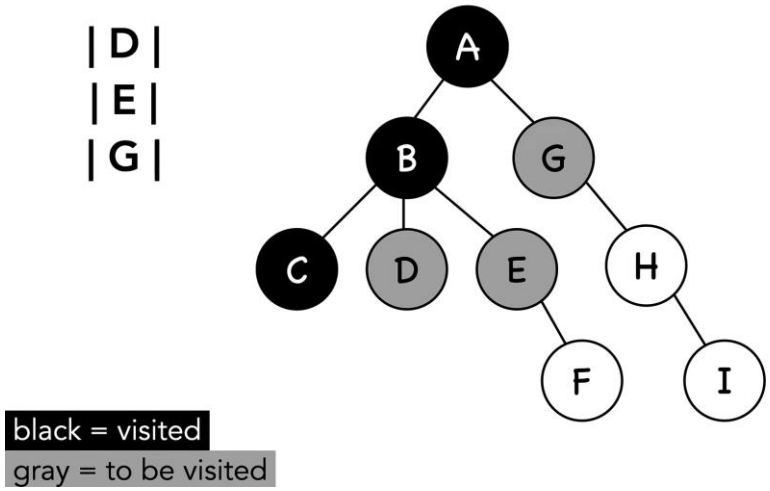

Рисунок 4 – Попали в узел C

У точки C нет потомков, следовательно возвращаемся в точку D, у которой также нет потомков, помечаем ей пройденной, убираем и её из стека и идём в точку E.

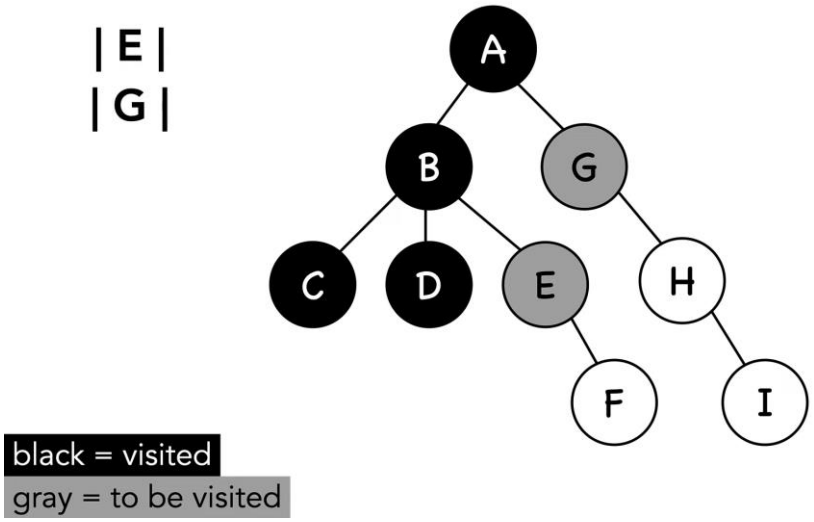

Рисунок 5 – Попали в узел D

Идём в узел E, удаляем его из стека и заносим его потомка F в вершину стека.

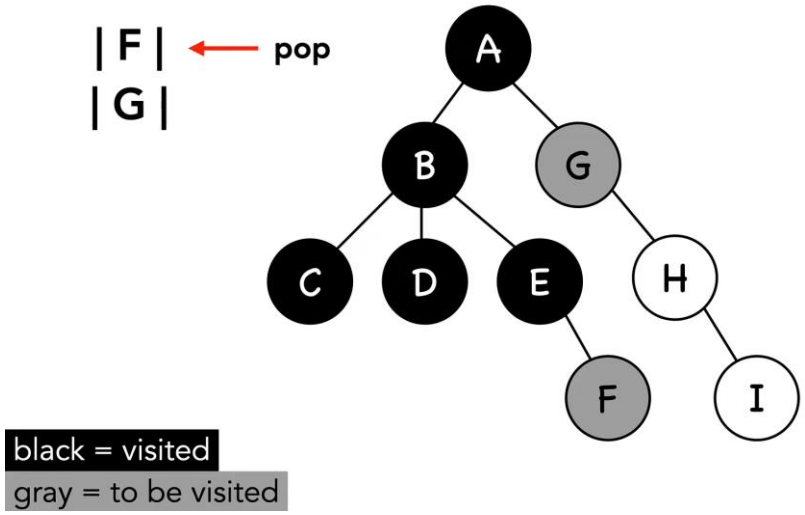

Рисунок 6 – Попали в узел E

Попадаем в узел F и убираем его из стека. В стеке осталась вершина G, с неё аналогично продолжиться обход графа по правой ветви.

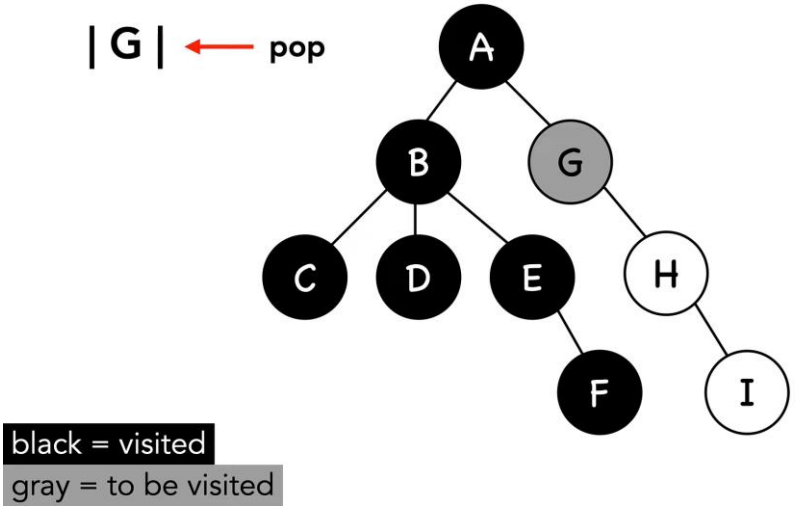

Рисунок 7 – Попали в узел F

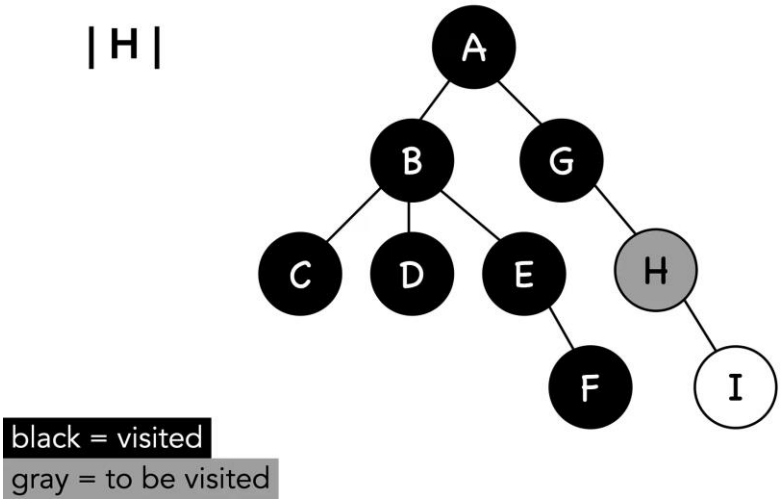

Рисунок 8 – Попали в узел G

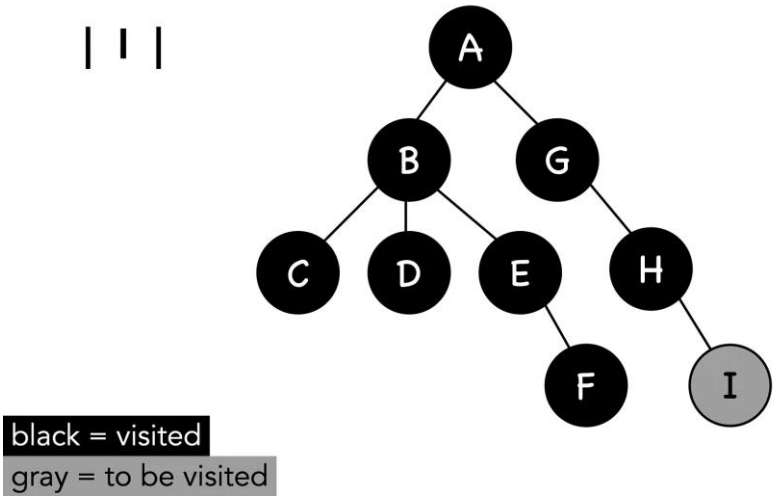

Рисунок 9 – Попали в узел H

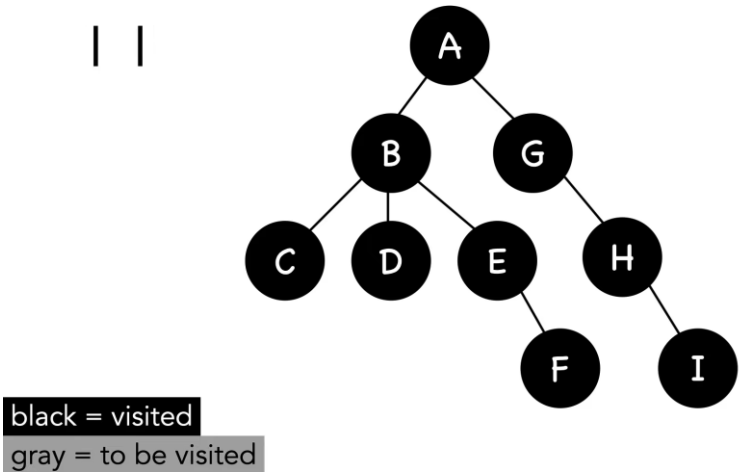

Рисунок 10 – Попали в узел I

Наконец, стек пуст и нам нечего в него положить, следовательно, алгоритм DFS закончен.

2. Обход в ширину (BFS)

На примере простого графа разберём принцип работы BFS более подробно. Пусть дан следующий граф. Начнём обход с узла A, и он сразу будет пройденным.

Узлы, окрашенные серым, находятся в очереди по принципу FIFO (англ. first in, first out).

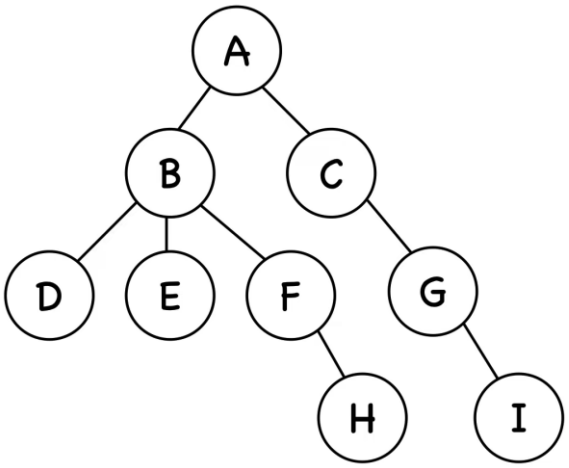

Рисунок 11 – Граф для BFS

Сразу пометим узел A, как пройденный. Занесём узлы B и C в очередь и удалим из неё узел B, как первый вошедший.

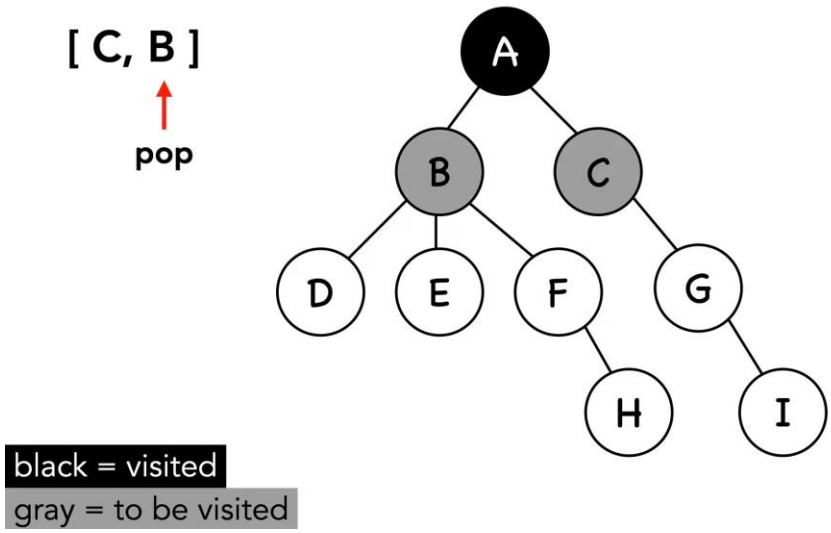

Рисунок 12 – Первый шаг BFS

Помечаем узел B, как пройденный и добавляем соседние с ним узлы D, E и F в очередь.

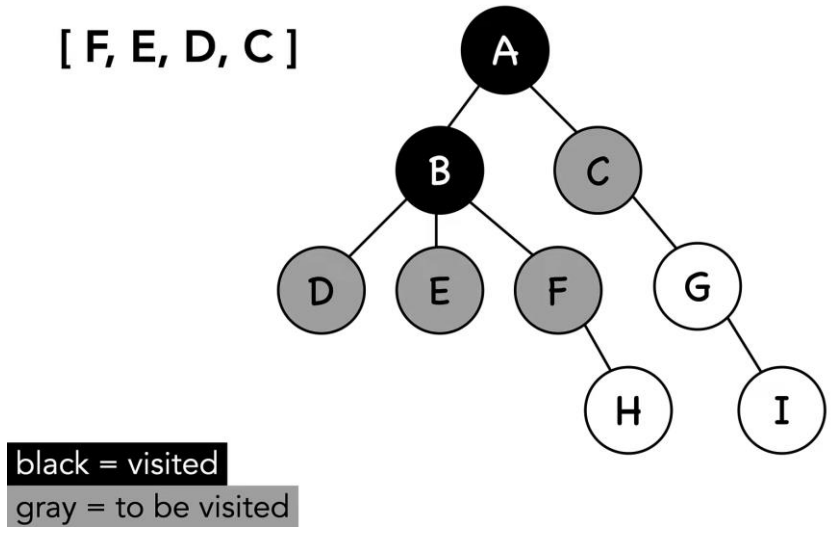

Рисунок 13 – Попали в узел B

Затем, перейдём в узел C, как в первый в очереди и занесём его потомок G в конец очереди. Тут мы можем наглядно увидеть почему этот называется поиском в ширину. Мы посетили все узлы в одном слое и только потом можем продвигаться дальше.

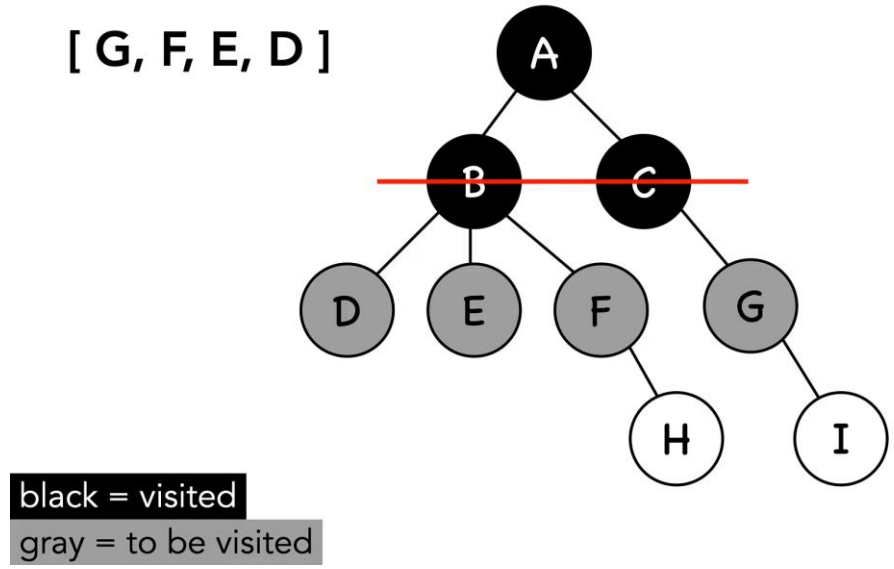

Рисунок 14 – Попали в узел C

Аналогичным образом мы посетим слой DEFG, оставив в очереди H и I.

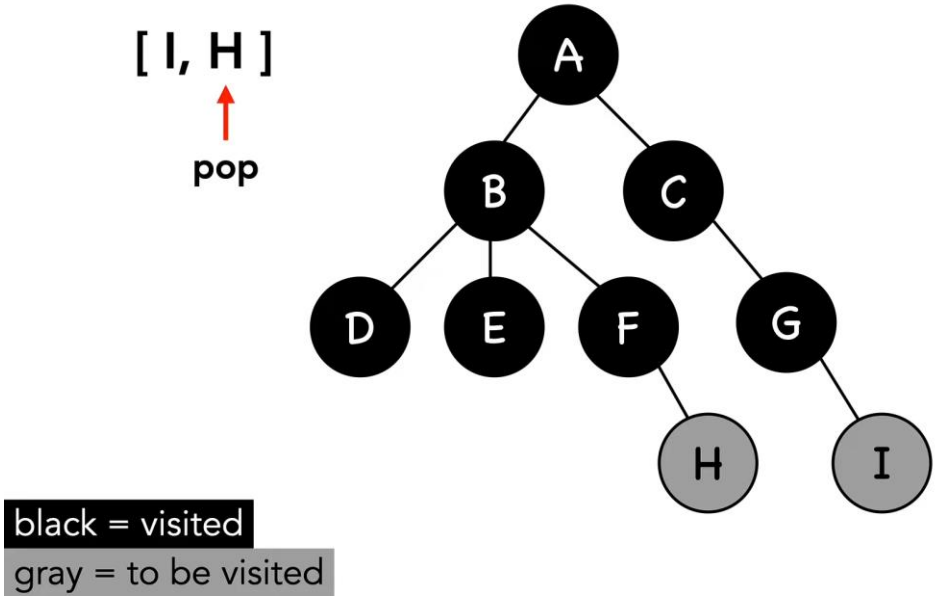

Рисунок 15 – Посетили слой DEFG

Наконец, посетим узлы H и I. В очереди ничего не осталось, следовательно, алгоритм BFS завершён.

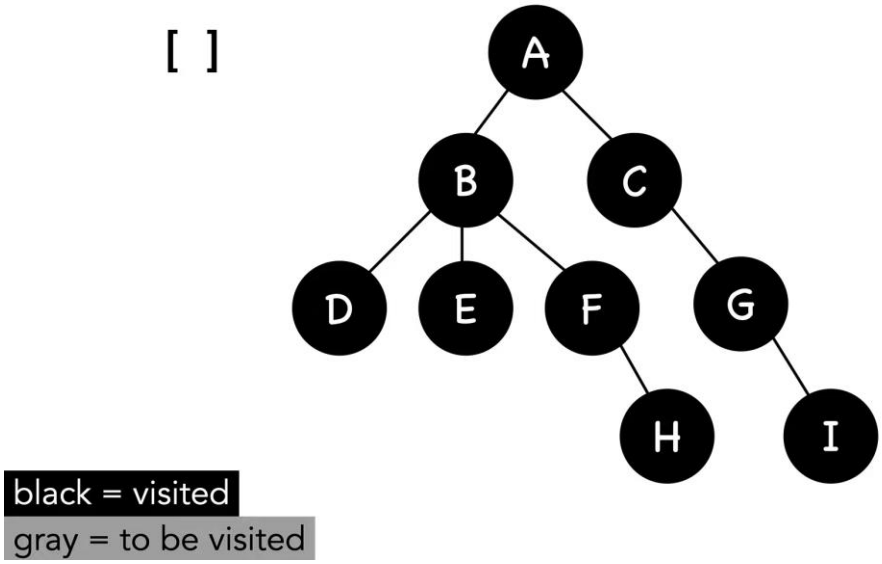

Рисунок 16 – Посетили слой HI

3. Топологическая сортировка (алгоритм Кана)

Рассмотрим принцип работы алгоритма Кана на примере простого графа.

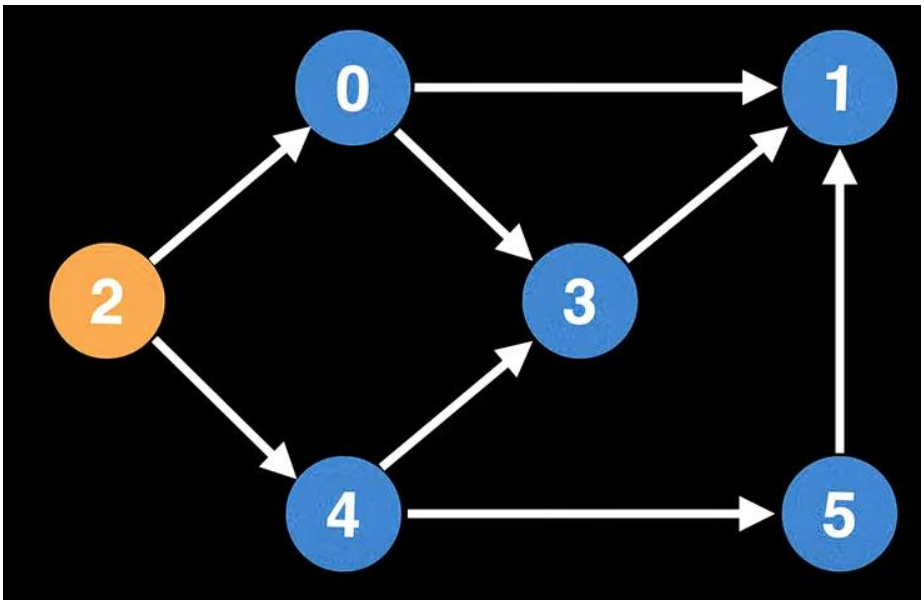

Рисунок 17 – Граф для алгоритма Кана

Тут нужно сделать 2 вещи: записать входящую степень каждого узла (количество входящих рёбер) и записывать в очередь узлы с входящей степенью равной нулю (эти узлы и будут посещаться на данном шаге).

Входящие степени для данного графа:

Узел	0	1	2	3	4	5
Степень	1	3	0	2	1	1

Узел 2 – единственный узел с нулевой степенью, поэтому с него и начнём обход, запишем первым его в топологический порядок и удалим из графа.

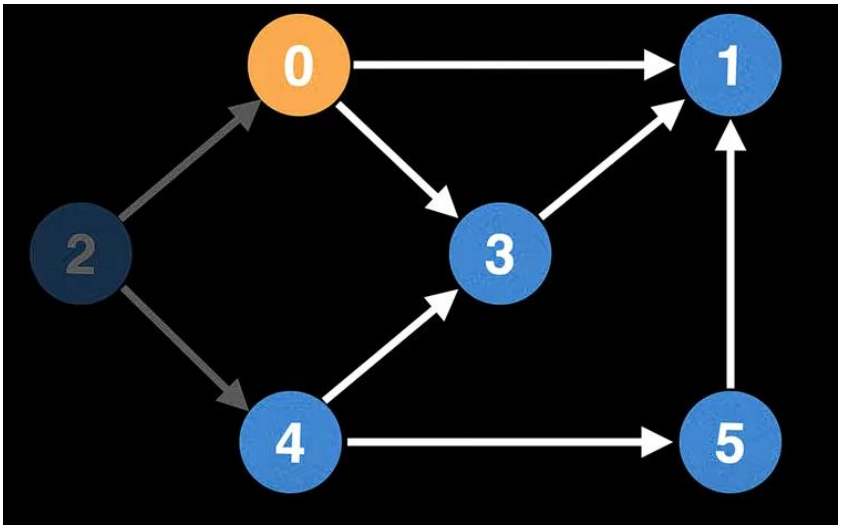

Рисунок 18 – Удалили узел 2

Узел	0	1	2	3	4	5
Степень	0	3	0	2	0	1

Теперь узлы 0 и 4 не имеют вхождений. Можем выбрать любой из них для записи в топологический порядок, пусть это будет 0. Удаляем его из графа. Аналогично поступим с узлом 4.

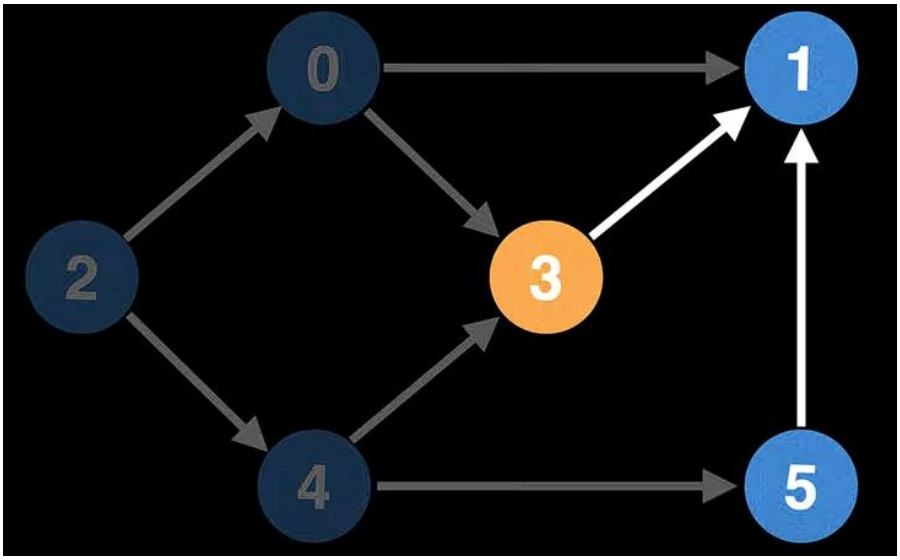

Рисунок 19 – Удалили узлы 0 и 4

Узел	0	1	2	3	4	5
Степень	0	2	0	0	0	0

Далее, узлы 3 и 5 больше не имеют зависимостей, запишем их в очередь и первым из неё достанем 3. Запишем 3 в топологический порядок и удалим из графа.

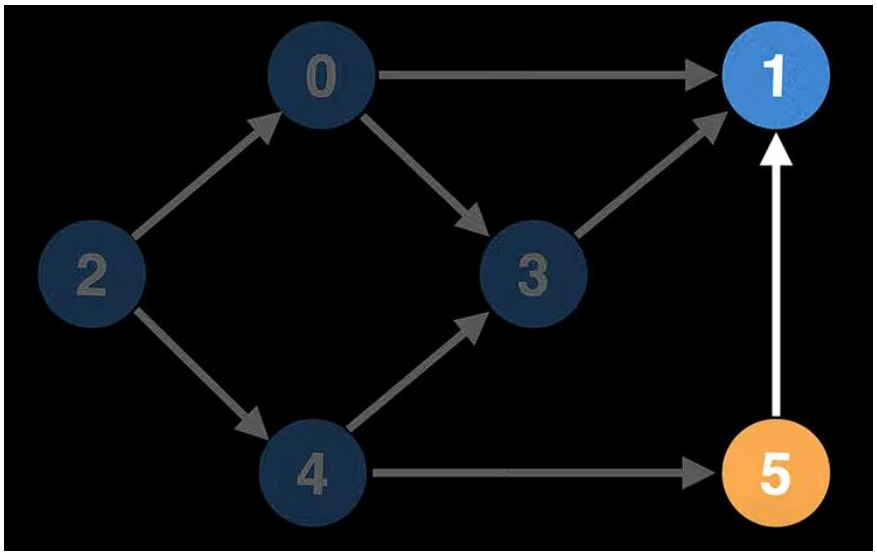

Рисунок 20 – Удалили узел 3

Аналогично достаём из очереди узел 5, записываем в топологический порядок и удаляем из графа.

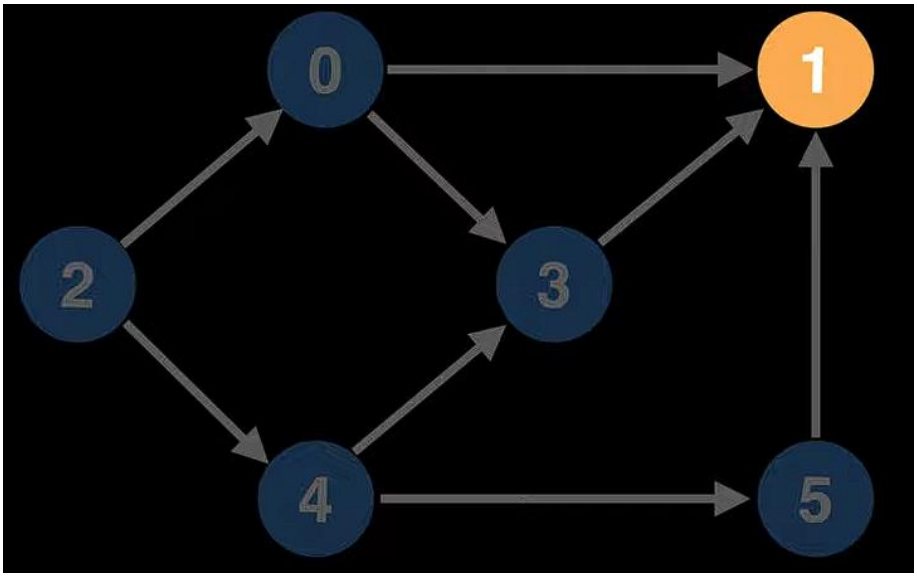

Рисунок 21 – Удалили узел 5

Наконец, остался только узел 1, запишем его в конец топологического порядка. На этом алгоритм Кана завершён.

Топологический порядок: 204351.

Реализуем на C++ данные алгоритмы для произвольного числа узлов и случайно заполним их данными.

Получившиеся коды программ представлены в Приложении 1.

На основе их тестирования построим графики зависимости времени выполнения алгоритмов (в мкс) от количества узлов в графе с помощью Excel.

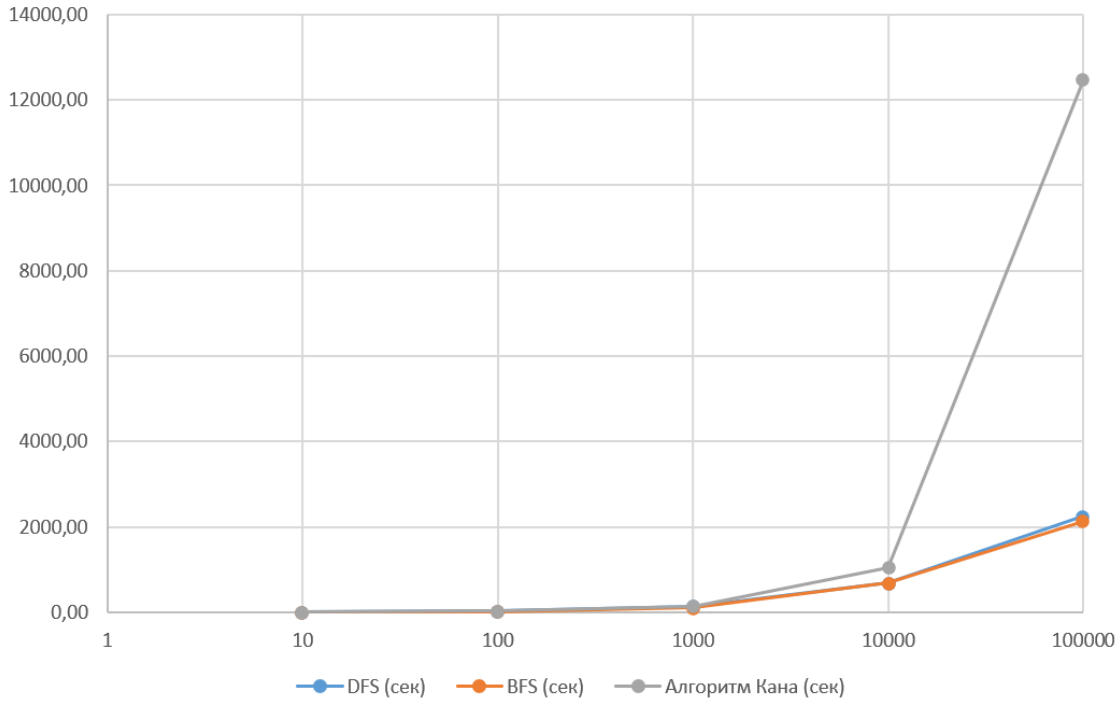

Рисунок 22 – Время выполнения программ в зависимости от количества узлов (Логарифмический масштаб)

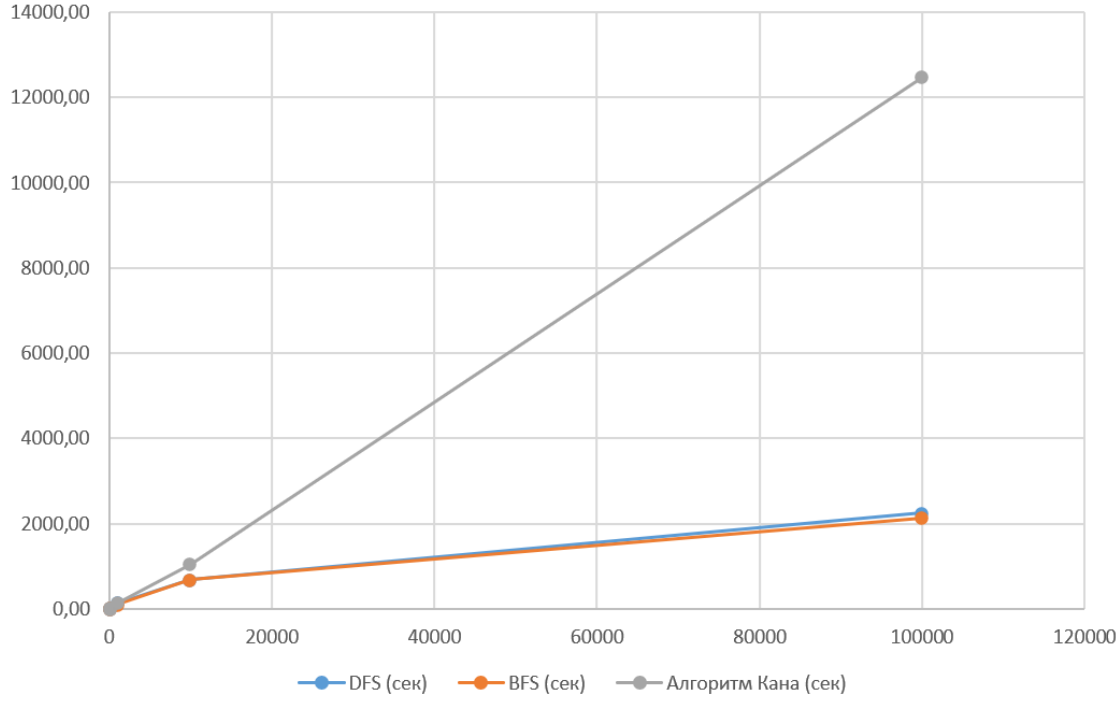

Рисунок 23 – Время выполнения программ в зависимости от количества узлов (Линейный масштаб)

Количество вершин	DFS (мкс)	BFS (мкс)	Алгоритм Кана (мкс)
10	7,05	5,16	4,80
100	29,88	20,98	29,84
1000	143,39	116,56	149,77
10000	694,21	685,46	1052,07
100000	2254,47	2137,16	12479,60

Как можно заметить, скорость выполнения алгоритмов DFS и BFS практически одинакова. И все три алгоритма имеют линейную зависимость времени выполнения от количества узлов.

Заключение

В ходе выполнения курсовой работы были успешно реализованы и проанализированы три ключевых алгоритма работы с графами: обход в глубину (DFS), обход в ширину (BFS) и топологическая сортировка (алгоритм Кана). Реализация на языке C++ продемонстрировала их эффективность и соответствие теоретическим ожиданиям. Экспериментальное тестирование и построение графиков подтвердили линейную зависимость времени выполнения алгоритмов от размера графа, что согласуется с их асимптотической сложностью O(V+E). Например, для графа с 100 000 вершин DFS и BFS показали время выполнения около 2,2 секунд, а алгоритм Кана — порядка 12.5 секунд, что подчеркивает их практическую применимость даже для крупных структур данных.

Практическая значимость работы заключается в возможности использования этих алгоритмов в реальных задачах, таких как анализ сетевых структур, планирование процессов с зависимостями и оптимизация маршрутов. Например, алгоритм Кана может быть интегрирован в системы управления проектами для определения корректного порядка выполнения задач, а DFS и BFS — в поисковые алгоритмы или анализ связности социальных сетей.

Список литературы

1. Поиск в глубину. Википедия. Режим доступа: https://ru.wikipedia.org/wiki/Поиск_в_глубину
2. Поиск в ширину. Википедия. Режим доступа: https://ru.wikipedia.org/wiki/Поиск_в_ширину
3. Кнут, Д.Э. Искусство программирования, Том 1. Основные алгоритмы, 3-е изд. : Пер. с англ. – М.: ООО «И.Д. Вильямс», 2018.
4. Kahn, A.B. Topological sorting of large networks. Communications of the ACM. — 1962. — Vol. 5. — P. 558–562.

Приложение 1

Обход графа в глубину (DFS):

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_set>
#include <random>
#include <chrono>
#include <stdexcept>

using namespace std;

// Проверка входных параметров
void validate_input(int num_nodes, int max_edges_per_node) {
    if (num_nodes <= 0) {
        throw invalid_argument("Количество вершин должно быть положительным.");
    }
    if (max_edges_per_node <= 0) {
        throw invalid_argument("Максимальное число рёбер на вершину должно быть
положительным.");
    }
}

vector<vector<int>> generate_dag(int num_nodes, int max_edges_per_node) {
    validate_input(num_nodes, max_edges_per_node); // Валидация входных данных

    vector<vector<int>> graph(num_nodes);
    mt19937 random_generator(1234);
    uniform_int_distribution<int> node_distribution(0, num_nodes - 1);

    try {
        for (int current_node = 0; current_node < num_nodes; ++current_node) {
            int max_possible_edges = min(max_edges_per_node, num_nodes - current_node - 1);
            if (max_possible_edges <= 0) continue;

            uniform_int_distribution<int> edge_count_distribution(0, max_possible_edges);
            int edge_count = edge_count_distribution(random_generator);

            for (int edge_index = 0; edge_index < edge_count; ++edge_index) {
                int target_node = current_node + 1 + (node_distribution(random_generator) % (num_nodes -
current_node - 1));
                if (target_node >= num_nodes) {
                    throw out_of_range("Некорректная генерация целевой вершины.");
                }
                graph[current_node].push_back(target_node);
            }
        }
    } catch (const exception& e) {
        cerr << "Ошибка генерации графа: " << e.what() << endl;
        throw;
    }

    return graph;
}
```

```
void dfs(const vector<vector<int>>& graph, int start_node) {
    if (start_node < 0 || start_node >= graph.size()) {
        throw out_of_range("Стартовая вершина вне диапазона графа.");
    }

    unordered_set<int> visited_nodes;
    stack<int> dfs_stack;
    dfs_stack.push(start_node);
    visited_nodes.insert(start_node);

    while (!dfs_stack.empty()) {
        int current_node = dfs_stack.top();
        dfs_stack.pop();

        for (auto neighbor_iterator = graph[current_node].rbegin(); neighbor_iterator !=
graph[current_node].rend(); ++neighbor_iterator) {
            int neighbor = *neighbor_iterator;
            if (neighbor < 0 || neighbor >= graph.size()) {
                throw out_of_range("Некорректная вершина в списке смежности.");
            }
            if (!visited_nodes.count(neighbor)) {
                visited_nodes.insert(neighbor);
                dfs_stack.push(neighbor);
            }
        }
    }
}

int main() {
    try {
        int num_nodes = 100000;
        int max_edges_per_node = 5;

        vector<vector<int>> graph = generate_dag(num_nodes, max_edges_per_node);

        int runs = 10;
        double total_time = 0;

        for (int run_index = 0; run_index < runs; ++run_index) {
            auto start_time = chrono::high_resolution_clock::now();
            dfs(graph, 0);
            auto end_time = chrono::high_resolution_clock::now();
            total_time += chrono::duration<double>(end_time - start_time).count();
        }

        cout << "Среднее время DFS: " << (total_time / runs) << " сек\n";
    } catch (const exception& e) {
        cerr << "Ошибка: " << e.what() << endl;
        return 1;
    }
    return 0;
}
```

Обход графа в ширину (BFS):

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
#include <random>
#include <chrono>
#include <stdexcept>

using namespace std;

void validate_input(int num_nodes, int max_edges_per_node) {
    if (num_nodes <= 0) {
        throw invalid_argument("Количество вершин должно быть положительным.");
    }
    if (max_edges_per_node <= 0) {
        throw invalid_argument("Максимальное число рёбер на вершину должно быть
положительным.");
    }
}

vector<vector<int>> generate_dag(int num_nodes, int max_edges_per_node) {
    validate_input(num_nodes, max_edges_per_node);

    vector<vector<int>> graph(num_nodes);
    mt19937 random_generator(1234);
    uniform_int_distribution<int> node_distribution(0, num_nodes - 1);

    try {
        for (int current_node = 0; current_node < num_nodes; ++current_node) {
            int max_possible_edges = min(max_edges_per_node, num_nodes - current_node - 1);
            if (max_possible_edges <= 0) continue;

            uniform_int_distribution<int> edge_count_distribution(0, max_possible_edges);
            int edge_count = edge_count_distribution(random_generator);

            for (int edge_index = 0; edge_index < edge_count; ++edge_index) {
                int target_node = current_node + 1 + (node_distribution(random_generator) % (num_nodes -
current_node - 1));
                if (target_node >= num_nodes) {
                    throw out_of_range("Некорректная генерация целевой вершины.");
                }
                graph[current_node].push_back(target_node);
            }
        }
    } catch (const exception& e) {
        cerr << "Ошибка генерации графа: " << e.what() << endl;
        throw;
    }

    return graph;
}

void bfs(const vector<vector<int>>& graph, int start_node) {
    if (start_node < 0 || start_node >= graph.size()) {
        throw out_of_range("Стартовая вершина вне диапазона графа.");
    }
```

```
    unordered_set<int> visited_nodes;
    queue<int> bfs_queue;
    bfs_queue.push(start_node);
    visited_nodes.insert(start_node);

    while (!bfs_queue.empty()) {
        int current_node = bfs_queue.front();
        bfs_queue.pop();

        for (int neighbor : graph[current_node]) {
            if (neighbor < 0 || neighbor >= graph.size()) {
                throw out_of_range("Некорректная вершина в списке смежности.");
            }
            if (!visited_nodes.count(neighbor)) {
                visited_nodes.insert(neighbor);
                bfs_queue.push(neighbor);
            }
        }
    }
}

int main() {
    try {
        int num_nodes = 100000;
        int max_edges_per_node = 5;

        vector<vector<int>> graph = generate_dag(num_nodes, max_edges_per_node);

        int runs = 10;
        double total_time = 0;

        for (int run_index = 0; run_index < runs; ++run_index) {
            auto start_time = chrono::high_resolution_clock::now();
            bfs(graph, 0);
            auto end_time = chrono::high_resolution_clock::now();
            total_time += chrono::duration<double>(end_time - start_time).count();
        }

        cout << "Среднее время BFS: " << (total_time / runs) << " сек\n";
    } catch (const exception& e) {
        cerr << "Ошибка: " << e.what() << endl;
        return 1;
    }
    return 0;
}
```

Топологическая сортировка (алгоритм Кана):

```
#include <iostream>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
#include <stdexcept>

using namespace std;

void validate_input(int num_nodes, int max_edges_per_node) {
    if (num_nodes <= 0) {
        throw invalid_argument("Количество вершин должно быть положительным.");
    }
    if (max_edges_per_node <= 0) {
        throw invalid_argument("Максимальное число рёбер на вершину должно быть
положительным.");
    }
}

vector<vector<int>> generate_dag(int num_nodes, int max_edges_per_node) {
    validate_input(num_nodes, max_edges_per_node);

    vector<vector<int>> graph(num_nodes);
    mt19937 random_generator(1234);
    uniform_int_distribution<int> node_distribution(0, num_nodes - 1);

    try {
        for (int current_node = 0; current_node < num_nodes; ++current_node) {
            int max_possible_edges = min(max_edges_per_node, num_nodes - current_node - 1);
            if (max_possible_edges <= 0) continue;

            uniform_int_distribution<int> edge_count_distribution(0, max_possible_edges);
            int edge_count = edge_count_distribution(random_generator);

            for (int edge_index = 0; edge_index < edge_count; ++edge_index) {
                int target_node = current_node + 1 + (node_distribution(random_generator) % (num_nodes -
current_node - 1));
                if (target_node >= num_nodes) {
                    throw out_of_range("Некорректная генерация целевой вершины.");
                }
                graph[current_node].push_back(target_node);
            }
        }
    } catch (const exception& e) {
        cerr << "Ошибка генерации графа: " << e.what() << endl;
        throw;
    }

    return graph;
}

vector<int> kahns(const vector<vector<int>>& graph) {
    if (graph.empty()) {
        throw invalid_argument("Граф пуст.");
    }
```

```
    int total_nodes = graph.size();
    vector<int> in_degree_count(total_nodes, 0);

    // Проверка корректности списка смежности
    for (const auto& neighbors : graph) {
        for (int target_node : neighbors) {
            if (target_node < 0 || target_node >= total_nodes) {
                throw out_of_range("Некорректная вершина в списке смежности.");
            }
            in_degree_count[target_node]++;
        }
    }

    queue<int> processing_queue;
    for (int node = 0; node < total_nodes; ++node) {
        if (in_degree_count[node] == 0) {
            processing_queue.push(node);
        }
    }

    vector<int> topological_order;
    while (!processing_queue.empty()) {
        int current_node = processing_queue.front();
        processing_queue.pop();
        topological_order.push_back(current_node);

        for (int neighbor : graph[current_node]) {
            if (--in_degree_count[neighbor] == 0) {
                processing_queue.push(neighbor);
            }
        }
    }

    if (topological_order.size() != total_nodes) {
        throw runtime_error("Граф содержит цикл.");
    }

    return topological_order;
}

int main() {
    try {
        int num_nodes = 100000;
        int max_edges_per_node = 5;

        vector<vector<int>> graph = generate_dag(num_nodes, max_edges_per_node);

        int runs = 10;
        double total_time = 0;

        for (int run_index = 0; run_index < runs; ++run_index) {
            auto start_time = chrono::high_resolution_clock::now();
            auto order = kahns(graph);
            auto end_time = chrono::high_resolution_clock::now();
            total_time += chrono::duration<double>(end_time - start_time).count();
        }
```

```
        cout << "Среднее время алгоритма Кана: " << (total_time / runs) << " сек\n";
    } catch (const exception& e) {
        cerr << "Ошибка: " << e.what() << endl;
        return 1;
    }
    return 0;
}
```