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»:

СТУДЕНТ группы КС-33 Костяева К.С.

Москва
2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ

Граф — это структура данных, состоящая из вершин (узлов) и рёбер (связей).

Графы бывают:

- **Ориентированные** (рёбра имеют направление) и **неориентированные** (связь в обе стороны).
- **Взвешенные** (у рёбер есть "стоимость", например, расстояние) и **невзвешенные**.
- **Разреженные** (мало рёбер) и **плотные** (почти все вершины соединены).

Поиск в ширину (BFS - Breadth-First Search):

Как работает:

- BFS использует **очередь (FIFO)**.
- Начинает с **стартовой вершины** и посещает все **соседние вершины**.
- Затем берёт **следующий уровень** вершин и повторяет процесс.

Алгоритм BFS:

1. Добавить начальную вершину в очередь.
2. Пока очередь **не пуста**:
    - Взять вершину из очереди.
    - Добавить **всех её соседей**, которые ещё не посещены.
3. Повторять, пока не обработаем весь граф.

Сложность алгоритма:

- **O(V + E)**, где V — вершины, E — рёбра.
- В худшем случае (плотный граф) **O(V²)**.

Поиск в глубину (DFS - Depth-First Search):

Как работает:

- DFS использует **стек (LIFO) или рекурсию**.
- Начинает с **стартовой вершины** и идёт вглубь по первому найденному пути.
- Если **упёрся в тупик**, возвращается назад и пробует другой путь.

Алгоритм DFS:

1. Добавить начальную вершину в стек.
2. Пока стек **не пуст**:
    - Взять вершину из стека.
    - Добавить **всех её соседей**, которых ещё не посещали.
3. Повторять, пока не посетим весь граф.

Сложность:

- **O(V + E)** (аналогично BFS).

ЗАДАНИЕ

========== Задание на лабораторную #4.

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.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

язык программирования C++

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

using namespace std;

class Graph {
private:
    int vertices;
    vector<vector<int>> adjacencyMatrix;
    vector<vector<int>> adjacencyList;
    vector<pair<int, int>> edgeList;
    bool directed;

public:
    Graph(int v, bool isDirected) : vertices(v), directed(isDirected) {
        adjacencyMatrix.resize(v, vector<int>(v, 0));
        adjacencyList.resize(v);
    }

    void addEdge(int u, int v) {
        adjacencyMatrix[u][v] = 1;
        adjacencyList[u].push_back(v);
        edgeList.emplace_back(u, v);
        if (!directed) {
            adjacencyMatrix[v][u] = 1;
            adjacencyList[v].push back(u);
            edgeList.emplace_back(v, u);
        }
    }

    vector<vector<int>> getAdjacencyMatrix() { return adjacencyMatrix; }
    vector<vector<int>> getAdjacencyList() { return adjacencyList; }
    vector<pair<int, int>> getEdgeList() { return edgeList; }

    bool bfs(int start, int end) {
        vector<bool> visited(vertices, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;
        while (!q.empty()) {
            int node = q.front(); q.pop();
            if (node == end) return true;
            for (int neighbor : adjacencyList[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        return false;
    }

    bool dfs(int start, int end) {
        vector<bool> visited(vertices, false);
        stack<int> s;
        s.push(start);
```

```
        while (!s.empty()) {
            int node = s.top(); s.pop();
            if (node == end) return true;
            if (!visited[node]) {
                visited[node] = true;
                for (int neighbor : adjacencyList[node]) {
                    if (!visited[neighbor]) {
                        s.push(neighbor);
                    }
                }
            }
        }
        return false;
    }
};

class GraphGenerator {
public:
    static Graph generateRandomGraph(int minVertices, int maxVertices, int minEdges, int
maxEdges,
        int maxEdgesPerVertex, bool directed) {
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> vertexDist(minVertices, maxVertices);
        uniform_int_distribution<int> edgeDist(minEdges, maxEdges);
        uniform_int_distribution<int> neighborDist(1, maxEdgesPerVertex);

        int vertices = vertexDist(gen);
        int edges = edgeDist(gen);
        Graph g(vertices, directed);

        set<pair<int, int>> edgeSet;
        for (int i = 0; i < edges; ++i) {
            int u = rand() % vertices;
            int v = rand() % vertices;
            if (u != v && edgeSet.find({ u, v }) == edgeSet.end() && edgeSet.find({ v, u
}) == edgeSet.end()) {
                g.addEdge(u, v);
                edgeSet.insert({ u, v });
            }
        }
        return g;
    }
};

void measurePerformance(int minVertices, int maxVertices, int minEdges, int maxEdges,
    int maxEdgesPerVertex, ofstream& outputFile) {
    Graph g = GraphGenerator::generateRandomGraph(minVertices, maxVertices, minEdges,
maxEdges,
        maxEdgesPerVertex, false);

    int start = rand() % g.getAdjacencyList().size();
    int end = rand() % g.getAdjacencyList().size();

    auto measureTime = [&](auto searchFunc) {
        auto start time = chrono::high resolution_clock::now();
        for (int i = 0; i < 1000; ++i) {
            searchFunc(start, end);
        }
        auto end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> elapsed = end_time - start_time;
        return elapsed.count() / 1000.0;
        };

    double bfsTime = measureTime([&](int a, int b) { return g.bfs(a, b); });
    double dfsTime = measureTime([&](int a, int b) { return g.dfs(a, b); });

    // Выводим в файл
```

```
    outputFile << g.getAdjacencyList().size() << "," << g.getEdgeList().size() << "," <<
bfsTime << "," << dfsTime << "\n";
    // Выводим в консоль
    cout << "Graph with " << g.getAdjacencyList().size() << " vertices and " <<
g.getEdgeList().size() << " edges.\n";
    cout << "BFS time: " << bfsTime << " s\n";
    cout << "DFS time: " << dfsTime << " s\n";
}

int main() {
    ofstream outputFile("graph_performance.csv"); // Открываем файл для записи
    outputFile << "Vertices,Edges,BFS_time,DFS_time\n"; // Заголовок CSV файла

    // Параметры для генерации графа
    int minVertices = 10;
    int maxVertices = 100;
    int minEdges = 10;
    int maxEdges = 200;
    int maxEdgesPerVertex = 10;

    for (int i = 0; i < 10; ++i) {
        measurePerformance(minVertices, maxVertices, minEdges, maxEdges,
maxEdgesPerVertex, outputFile);
        minVertices += 10;
        maxVertices += 10;
    }

    outputFile.close(); // Закрываем файл
    return 0;
}
```

Результат работы программы:

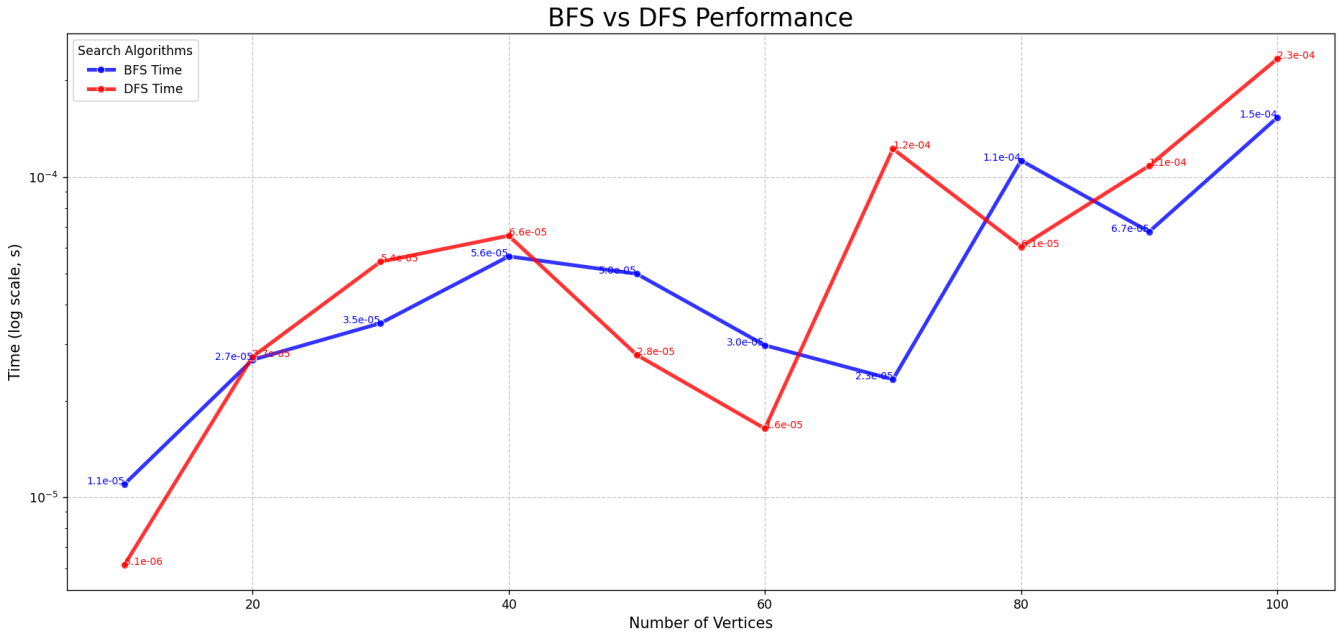
```
Graph with 10 vertices and 10 edges:
Adjacency Matrix:
 0  0  0  0  1  0  0  0  0  0
 0  0  0  0  1  0  1  1  0  0
 0  0  0  1  1  1  0  0  0  0
 0  0  1  0  0  0  0  0  0  0
 1  1  1  0  0  0  0  0  0  1
 0  0  1  0  0  0  0  0  1  0
 0  1  0  0  0  0  0  1  0  0
 0  1  0  0  0  0  1  0  0  0
 0  0  0  0  0  1  0  0  0  0
 0  0  0  0  1  0  0  0  0  0

Incidence Matrix:
 0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  1  0  0  0  0  0  0  0  0  0  0  1  1  0  0  1  1  0  0
 0  0  0  0  0  0  1  1  1  1  0  0  0  0  1  1  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0
 0  0  1  1  1  1  1  1  0  0  0  0  1  1  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  1  1
 0  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  1  1  0  0
 1  1  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
 0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0

BFS time: 1.0981e-05 s
DFS time: 6.1452e-06 s
```

Графики:

язык программирования Python



Ось X — количество вершин.
Ось Y — время выполнения (в секундах).
Два разных цвета (синий для BFS и красный для DFS) показывает, как время работы этих алгоритмов изменяется в зависимости от размера графа

ВЫВОД

В ходе данной работы мы изучили и сравнили два основных алгоритма обхода графов: **поиск в ширину (BFS)** и **поиск в глубину (DFS)**. Построили графики времени выполнения этих алгоритмов в зависимости от количества вершин в графе и провели анализ их производительности.

- **BFS** лучше подходит для поиска кратчайших путей и равномерного обхода графа.
- **DFS** может быть быстрее в некоторых случаях, особенно в задачах, требующих глубокого погружения в структуру графа.
- Производительность алгоритмов зависит от **количества вершин, рёбер и структуры графа**.
- Время выполнения BFS и DFS может **существенно варьироваться**, особенно на графах с разной плотностью.