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-30..... Лихолат Полина Николаевна
Ссылка на репозиторий: .....https://github.com/MUCTR-IKT-CPP/Likholat_algorithms.git

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: 20.02.2023

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 метода/модели.

В коде реализован алгоритм поиска в ширину (BFS) для нахождения кратчайшего пути между двумя вершинами, исходной и целевой, в графе, представленном матрицей смежности.

Алгоритм работает следующим образом:

1. Инициализируйте вектор с именем "посещенный", чтобы отслеживать посещенные вершины. Инициализируйте все записи значением 0.

2. Инициализируйте вектор с именем "parent", чтобы отслеживать родительский элемент каждой вершины по кратчайшему пути. Инициализируйте все записи значением -1.
3. Инициализируйте очередь с именем "q" и поместите в нее исходную вершину.
4. Отметьте исходную вершину как посещенную, установив для соответствующей записи в векторе посещений значение 1.
5. Пока очередь не пуста:
    - Удалите из очереди вершину "u" из начала очереди.
    - Для каждой соседней вершины "v" из "u":
    - Если "v" не был посещен и есть граница между "u" и "v":
6. Отметьте "v" как посещенный.
7. Установите родительское значение "v" на "u".
8. Поставьте в очередь "v".

Если целевая вершина была посещена, строится кратчайший путь, следуя родительским указателям, начиная с целевой вершины, пока не будет достигнута исходная вершина. Путь сохраняется в векторе с именем "path".

Порядок вершин в векторе пути изменяется в обратном порядке.

Возвращается вектор пути, если он существует (т.е. была посещена целевая вершина), в противном случае возвращается пустой вектор.

В целом, алгоритм исследует граф вширь, посещая все вершины на расстоянии k от источника, прежде чем перейти к вершинам на расстоянии k + 1. Родительские указатели помогают восстановить кратчайший путь от целевой вершины к исходной вершине. Временная сложность этого алгоритма равна O(V+E), где V - количество вершин, а E - количество ребер в графе.

Также мы используем алгоритм поиска в глубину (DFS) для нахождения кратчайшего пути между двумя вершинами, исходной и целевой, в графе, представленном матрицей смежности.

Алгоритм работает следующим образом:

1. Инициализируется вектор с именем "visited", чтобы отслеживать посещенные вершины. Инициализируем все записи значением 0.
2. Инициализируется вектор с именем "parent", чтобы отслеживать родительский элемент каждой вершины по кратчайшему пути. Инициализируем все записи значением -1.
3. Инициализируем стек с именем "s" и вставляем в него исходную вершину.
4. Отмечаем исходную вершину как посещенную, установив для соответствующей записи в векторе посещений значение 1.

5. Пока стек не пуст:
    a. Извлекаем вершину "u" из верхней части стека.
    b. Для каждой смежной вершины "v" из "u": если "v" не был посещен и есть граница между "u" и "v":
        - Устанавливаем родительское значение "v" на "u".
        - Вставляем "v" в стек.
        - Если "v" является целевой вершиной, выходим из цикла.
6. Если целевая вершина была посещена, строим кратчайший путь, следуя родительским указателям, начиная с целевой вершины, пока не будет достигнута исходная вершина. Сохраняем путь в векторе с именем "path".
7. Изменяем порядок вершин в векторе пути в обратном порядке.
8. Возвращается вектор пути, если он существует (т.е. была посещена целевая вершина), в противном случае возвращается пустой вектор.

В целом, алгоритм исследует граф в первую очередь в глубину, посещая вершины вдоль пути, пока он не достигнет тупика, а затем возвращается к самой последней вершине с неисследованными соседями. Родительские указатели помогают восстановить кратчайший путь от целевой вершины к исходной вершине. Временная сложность этого алгоритма равна O(V+E), где V - количество вершин, а E - количество ребер в графе. Однако, важно отметить, что DFS не обязательно находит кратчайший путь в невзвешенном графе, в отличие от BFS.

Выполнение задачи.

Сначала подключаем необходимые библиотеки:

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <fstream>

using namespace std;
```

Затем создадим класс графа, объявим необходимые переменные и прототипы функций:

```
class Graph {
public:
    Graph(int vertices, bool directed = false);
    void add_edge(int u, int v, int weight = 1);
    vector<vector<int>> get_adj_matrix();
    vector<vector<pair<int, int>>> get_adj_list();
    vector<pair<int, int>> get_edges();
    vector<vector<int>> get_inc_matrix();
    void print_adj_matrix() const;
    void print_inc_matrix() const;
    void print_adj_list() const;
    void print_edge_list() const;
```

```
private:
    int vertices_;
    bool directed_;
    vector<vector<int>> adj_matrix_;
    vector<vector<pair<int, int>>> adj_list_;
    vector<pair<int, int>> edges_;
    vector<vector<int>> inc_matrix_;
};
```

Опишем объявленные методы класса граф:

```
Graph::Graph(int vertices, bool directed) {
    vertices_ = vertices;
    directed_ = directed;
    adj_matrix_ = vector<vector<int>>(vertices, vector<int>(vertices, 0));
    adj_list_ = vector<vector<pair<int, int>>>(vertices);
}

void Graph::add_edge(int u, int v, int weight) {
    adj_matrix_[u][v] = weight;
    adj_list_[u].push_back(make_pair(v, weight));
    edges_.push_back(make_pair(u, v));
    if (directed_) {
        adj_matrix_[u][v] = weight;
    }
    else {
        adj_matrix_[u][v] = adj_matrix_[v][u] = weight;
    }
}

vector<vector<int>> Graph::get_adj_matrix() {
    return adj_matrix_;
}

vector<vector<pair<int, int>>> Graph::get_adj_list() {
    return adj_list_;
}

vector<pair<int, int>> Graph::get_edges() {
    return edges_;
}

vector<vector<int>> Graph::get_inc_matrix() {
    if (inc_matrix_.empty()) {
        inc_matrix_              =              vector<vector<int>>(vertices_,
vector<int>(edges_.size(), 0));
        for (int i = 0; i < edges_.size(); i++) {
            int u = edges_[i].first;
            int v = edges_[i].second;
            int weight = abs(adj_matrix_[u][v]);
            inc_matrix_[u][i] = weight;
            inc_matrix_[v][i] = directed_ ? -weight : weight;
        }
    }
    return inc_matrix_;
}

void Graph::print_adj_matrix() const {
    cout << "  ";
    for (int i = 0; i < vertices_; i++) { cout << "V" << i << " "; }
    cout << endl;
    for (int i = 0; i < vertices_; i++) {
```

```
        cout << "V" << i << " ";
        for (int j = 0; j < vertices_; j++) {
            /*cout << adj_matrix_[i][j] << " ";*/
            if (adj_matrix_[i][j] > 0) cout << 1 << "  ";
            else cout << 0 << "  ";
        }
        cout << endl;
    }
    cout << endl;
}

void Graph::print_inc_matrix() const {
    vector<vector<int>> inc_matrix(edges_.size(), vector<int>(vertices_, 0));
    int edge_idx = 0;
    for (int i = 0; i < vertices_; i++) {
        for (int j = i + 1; j < vertices_; j++) {
            if (adj_matrix_[i][j] != 0) {
                inc_matrix[edge_idx][i] = adj_matrix_[i][j];
                inc_matrix[edge_idx][j] = -adj_matrix_[i][j];
                edge_idx++;
            }
        }
    }
    cout << "  ";
    for (int i = 0; i < vertices_; i++) { cout << "V" << i << " "; }
    cout << endl;
    for (int i = 0; i < edges_.size(); i++) {
        cout << "E" << i << " ";
        for (int j = 0; j < vertices_; j++) {
            /*cout << inc_matrix[i][j] << " ";*/
            if (inc_matrix[i][j] != 0) cout << 1 << "  ";
            else cout << 0 << "  ";
        }
        cout << endl;
    }
    cout << endl;
}

void Graph::print_adj_list() const {
    for (int i = 0; i < vertices_; i++) {
        cout << i << ": ";
        for (int j = 0; j < vertices_; j++) {
            if (adj_matrix_[i][j] != 0) {
                cout << j << "(" << adj_matrix_[i][j] << ") ";
            }
        }
        cout << endl;
    }
    cout << endl;
}

void Graph::print_edge_list() const {
    for (auto edge : edges_) {
        cout << edge.first << " -> " << edge.second;
        if (adj_matrix_[edge.first][edge.second] != 0) {
            cout << " (" << adj_matrix_[edge.first][edge.second] << ")";
        }
        cout << endl;
    }
    cout << endl;
}
```

Затем создаём функцию генерации графа:

```
Graph generate_graph(int min_vertices, int max_vertices, int min_edges, int max_edges, int max_edges_per_vertex, bool directed, int max_incoming_edges, int max_outgoing_edges) {
    int num_vertices = rand() % (max_vertices - min_vertices + 1) + min_vertices;
    Graph g(num_vertices);
    int max_possible_edges = num_vertices * (num_vertices - 1) / 2;
    int max_possible_edges_per_vertex = num_vertices - 1;

    int num_edges = rand() % (max_edges - min_edges + 1) + min_edges;
    vector<int> edges_per_vertex(num_vertices, 0);
    vector<int> incoming_edges_per_vertex(num_vertices, 0);
    vector<int> outgoing_edges_per_vertex(num_vertices, 0);

    while (num_edges > 0) {
        int u = rand() % num_vertices;
        int v = rand() % num_vertices;
        if (u == v) {
            continue;
        }

        if (edges_per_vertex[u] >= max_possible_edges_per_vertex || edges_per_vertex[v] >= max_possible_edges_per_vertex) {
            continue;
        }

        if (directed && (incoming_edges_per_vertex[v] >= max_incoming_edges || outgoing_edges_per_vertex[u] >= max_outgoing_edges)) {
            continue;
        }

        if (g.get_adj_matrix()[u][v] != 0) {
            continue;
        }

        int weight = rand() % 100 + 1;
        g.add_edge(u, v, weight);
        num_edges--;
        edges_per_vertex[u]++;
        edges_per_vertex[v]++;

        if (directed) {
            incoming_edges_per_vertex[v]++;
            outgoing_edges_per_vertex[u]++;
        }

        if (num_edges == 0) {
            break;
        }
    }

    return g;
}
```

Создадим функцию для реализации поиска кратчайшего пути в ширину:

```
vector<int> bfs_shortest_path(Graph g, int source, int target) {
    int num_vertices = g.get_adj_matrix().size();
    vector<int> visited(num_vertices, 0);
```

```
    vector<int> parent(num_vertices, -1);
    queue<int> q;
    q.push(source);
    visited[source] = 1;

    while (!q.empty()) {
        int u = q.front();
        //if (u == target) break;
        q.pop();
        for (int v = 0; v < num_vertices; v++) {
            if (g.get_adj_matrix()[u][v] != 0 && !visited[v]) {
                visited[v] = 1;
                parent[v] = u;
                q.push(v);
            }
        }

    }

    if (visited[target]) {
        vector<int> path;
        int u = target;
        while (u != -1) {
            path.push_back(u);
            u = parent[u];
        }
        reverse(path.begin(), path.end());
        return path;
    }
    else {
        return vector<int>();
    }
}
```

Создадим функцию для реализации поиска кратчайшего пути в глубину:

```
vector<int> dfs_shortest_path(Graph g, int source, int target) {
    int num_vertices = g.get_adj_matrix().size();
    vector<int> visited(num_vertices, 0);
    vector<int> parent(num_vertices, -1);
    stack<int> s;
    s.push(source);
    visited[source] = 1;

    while (!s.empty()) {
        int u = s.top();
        s.pop();
        for (int v = 0; v < num_vertices; v++) {
            if (g.get_adj_matrix()[u][v] != 0 && !visited[v]) {
                visited[v] = 1;
                parent[v] = u;
                s.push(v);
                if (v == target) {
                    break;
                }
                /*u = s.top();*/
            }
        }
        /*s.pop();*/
    }
    if (visited[target]) {
        vector<int> path;
```

```
            int u = target;
            while (u != -1) {
                path.push_back(u);
                u = parent[u];
            }
            reverse(path.begin(), path.end());
            return path;
        }
        else {
            return vector<int>();
        }
    }
```

Проведем тесты и выведем результаты выполнения программы в консоль (рис.1):

```
    int main() {
        srand(time(0));
        int min_vertices = 100;
        int max_vertices = 100;
        int min_edges = 200;
        int max_edges = 200;
        int max_edges_per_vertex = 1;
        int num_graphs = 10;
        int source = 0;
        int target = 0;

        ofstream fout("output.txt");

        for (int i = 0; i < num_graphs; i++) {

            int num_vertices = min_vertices + i;
            int num_edges = min_edges + i * 2;
            Graph g = generate_graph(num_vertices, num_vertices, num_edges,
num_edges, max_edges_per_vertex, false, 0, 0);
            source = rand() % num_vertices;
            target = rand() % num_vertices;



            cout << "Adjacency matrix:" << endl;
            g.print_adj_matrix();

            cout << "Incidence matrix:" << endl;
            g.print_inc_matrix();

            cout << "Adjacency list:" << endl;
            g.print_adj_list();

            cout << "Edge list:" << endl;
            g.print_edge_list();

            cout << "Graph " << i + 1 << " with " << num_vertices << " vertices
and " << num_edges << " edges" << endl;
            fout << "Graph " << i + 1 << " with " << num_vertices << " vertices
and " << num_edges << " edges" << endl;
            clock_t start_bfs = clock();
            vector<int> bfs_path = bfs_shortest_path(g, source, target);
            clock_t end_bfs = clock();
            double time_bfs = (double)(end_bfs - start_bfs) / CLOCKS_PER_SEC;

            cout << "BFS shortest path from vertex " << source << " to vertex "
<< target << ": ";
```

```
            fout << "BFS shortest path from vertex " << source << " to vertex "
<< target << ": ";
            if (!bfs_path.empty()) {
                for (int j = 0; j < bfs_path.size(); j++) {
                    cout << bfs_path[j] << " ";
                    fout << bfs_path[j] << " ";
                }
                cout << endl;
                fout << endl;
            }
            else {
                cout << "Path does not exist" << endl;
                fout << "Path does not exist" << endl;
            }
            cout << "BFS shortest path time: " << time_bfs << " seconds" << endl;
            fout << "BFS shortest path time: " << time_bfs << " seconds" << endl;

            clock_t start_dfs = clock();
            vector<int> dfs_path = dfs_shortest_path(g, source, target);
            clock_t end_dfs = clock();
            double time_dfs = (double)(end_dfs - start_dfs) / CLOCKS_PER_SEC;

            cout << "DFS shortest path from vertex " << source << " to vertex "
<< target << ": ";
            fout << "DFS shortest path from vertex " << source << " to vertex "
<< target << ": ";
            if (!dfs_path.empty()) {
                for (int j = 0; j < dfs_path.size(); j++) {
                    cout << dfs_path[j] << " ";
                    fout << dfs_path[j] << " ";
                }
                cout << endl;
                fout << endl;
            }
            else {
                cout << "Path does not exist" << endl;
                fout << "Path does not exist" << endl;
            }
            cout << "DFS shortest path time: " << time_dfs << " seconds ";
            fout << "DFS shortest path time: " << time_dfs << " seconds ";
            cout << endl << endl << endl;
            fout << endl << endl << endl;

        }
        fout.close();
        return EXIT_SUCCESS;
    }
```

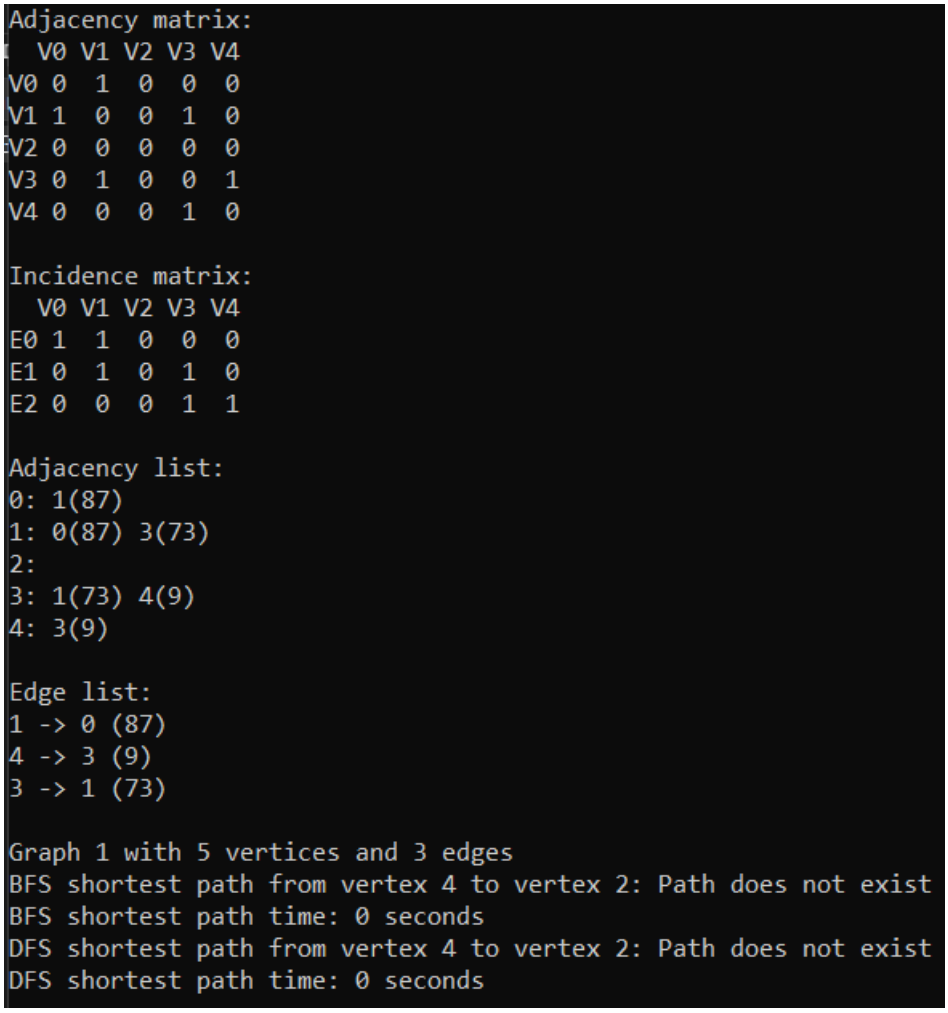

Рисунок 1. Пример вывода программы

По полученным результатам строим графики с помощью Python.

```
import matplotlib.pyplot as plt

# Времена BFS и DFS для каждого графа
bfs_times = [2.966, 2.835, 2.88, 2.962, 3.03, 3.019, 3.241, 3.264, 3.281, 3.28]
dfs_times = [2.695, 2.816, 2.845, 2.969, 2.982, 3.013, 3.163, 3.134, 3.193, 3.176]

# Номера графов
graph_numbers = list(range(1, 11))

# Построение графиков
plt.plot(graph_numbers, bfs_times, label='BFS')
plt.plot(graph_numbers, dfs_times, label='DFS')
plt.xlabel('Номер графа')
plt.ylabel('Время (секунды)')
plt.title('Зависимость времени обходов от номера графа')
plt.legend()
plt.show()
```

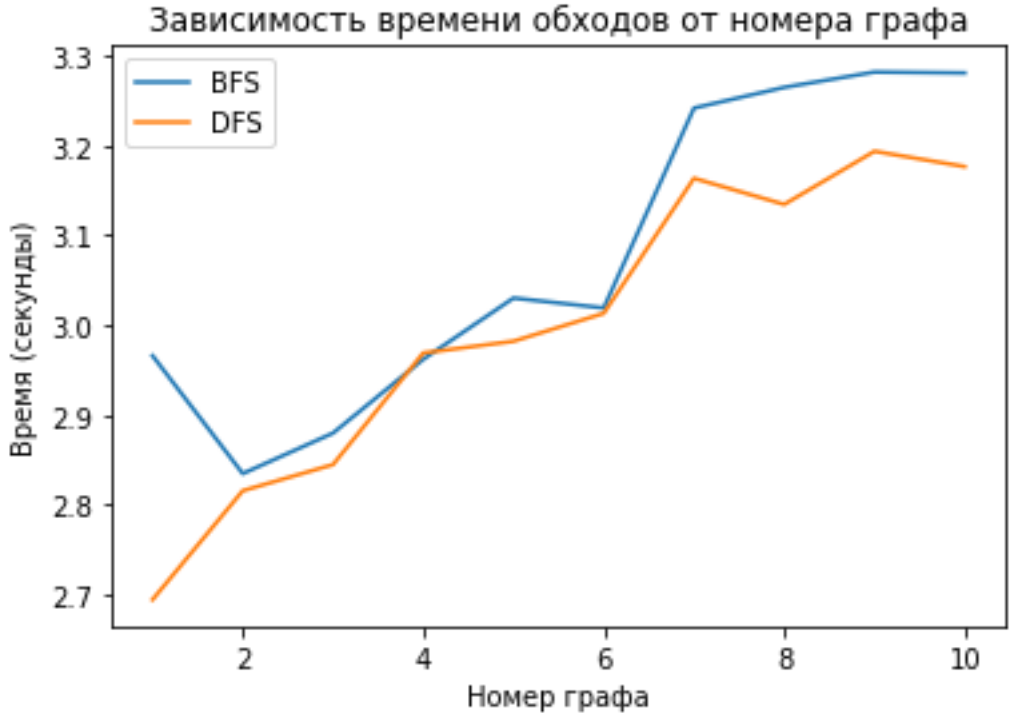

Рисунок 2. Зависимость времени обходов от номера графа

Заключение.

Алгоритм поиска кратчайшего пути в графе является одной из основных задач в теории графов и находит множество применений в различных областях, таких как транспортное планирование, сетевое проектирование, маршрутизация сетей и другие.

BFS (breadth-first search) и DFS (depth-first search) являются двумя основными алгоритмами поиска кратчайшего пути в графе. Каждый из них имеет свои преимущества и недостатки.

Преимущества BFS:

- Гарантирует нахождение кратчайшего пути.
- Работает оптимально на графах, где пути имеют мало ветвлений.
- Может использоваться для поиска всех путей между двумя вершинами.
- Имеет простую реализацию.

Недостатки BFS:

- Может быть неэффективен на графах с большим количеством ребер.
- Не подходит для поиска кратчайшего пути на графах с отрицательными весами ребер (в таком случае используются алгоритмы, основанные на поиске кратчайших путей в ациклических графах).

Преимущества DFS:

- Эффективен на графах с большим количеством ребер и малым количеством уровней.
- Имеет простую реализацию.
- Может использоваться для поиска всех путей между двумя вершинами.

Недостатки DFS:

- Не гарантирует нахождение кратчайшего пути.
- Может зациклиться на графах с циклами.

Таким образом, выбор между BFS и DFS зависит от характеристик графа и требований к решению задачи. Если нужно найти кратчайший путь в графе с малым количеством ребер и ветвлениями, лучше использовать BFS. Если граф имеет много ребер и небольшое количество уровней, DFS может оказаться более эффективным.