**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«Российский университет дружбы народов имени Патриса Лумумбы»**

Инженерная академия

*Кафедра механики и процессов управления*

**Курсовая работа**

По информатике и программированию

**Направление:** Прикладная математика и информатика

**Профиль:** Математические методы механики космического полёта и анализ геоинформационных данных

**Тема: Графическая визуализация алгоритмов на графах: DFS**

Выполнено студентами: Никитина Анфиса Геннадьевна

Лаптева Марина Григорьевна

Группа: ИПМбд-01-23

№ студенческого: 1132233505

1132233514

**Москва, 2025**

**Вступление**

**Граф** — это структура данных, состоящая из множества вершин (узлов) и рёбер (связей) между ними. Графы широко используются для моделирования сетей, связей и маршрутов. В нашем случае граф представлен списком смежности, где каждая вершина знает своих соседей.

Об алгоритме обхода в глубину (DFS)

**Обход в глубину (Depth-First Search, DFS)** — алгоритм для обхода или поиска в графе. Он начинает с выбранной вершины и исследует как можно дальше по каждой ветви перед возвратом назад. Это реализуется с помощью стека (рекурсия или явный стек).

**Цель работы**

Что делает эта программа?

- Визуализирует граф в консоли, отображая вершины и рёбра с помощью ASCII-символов.

- Показывает процесс обхода графа в глубину, выделяя посещённые вершины символом [\*].

- Делает паузу после посещения каждой вершины, чтобы наглядно видеть ход алгоритма.

**Решение**

Как это работает в коде?

1. Граф и позиции:

Граф задан как map<int, vector<int>>, где ключ — вершина, а значение — список соседей. Для каждой вершины указаны координаты на "канвасе" — двумерном массиве символов.

2. Рисование графа:

Функции drawVertex и drawEdge рисуют вершины и рёбра на текстовом канвасе. Вершина отображается как [n] или [\*], если посещена. Рёбра рисуются линиями -, |, /, \\.

3. Обход в глубину (DFS):

Функция dfs использует стек для обхода. После посещения каждой вершины происходит обновление канваса и вывод его на экран. Затем пауза для визуального восприятия.

Как запустить

windows:

g++ -std=c++11 -pthread graph\_dfs.cpp -o graph\_dfs.exe

graph\_dfs.exe

linux:

g++ -std=c++11 -pthread graph\_dfs.cpp -o graph\_dfs

./graph\_dfs

**КОД**

#include <iostream>

#include <map>

#include <vector>

#include <stack>

#include <set>

#include <thread>

#include <chrono>

#include <string>

#include <algorithm>

using std::cout;

using std::endl;

using std::map;

using std::pair;

using std::set;

using std::stack;

using std::string;

using std::to\_string;

using std::vector;

using std::min;

using std::max;

using std::this\_thread::sleep\_for;

using std::chrono::milliseconds;

const int CANVAS\_HEIGHT = 9; // высота холста

const int CANVAS\_WIDTH = 25; // ширина холста

// Граф (создаем мапу с ключем - вершиной и значением - списком связанные с ним вершины) в котором у вершины 1 связи с 5 и 6, 2 вершины связи с 3, 4, 5)

map<int, vector<int>> graph = {

{1, {5, 6}},

{2, {3, 4, 5}},

{3, {2}},

{4, {2, 5, 6}},

{5, {1, 2, 4}},

{6, {1, 4}}

};

// Позиции на экране

map<int, pair<int, int>> positions = {

{1, {4, 4}},

{2, {2, 14}},

{3, {0, 14}},

{4, {2, 9}},

{5, {4, 14}},

{6, {7, 9}}

};

// Холст (создаем пустой канвас на котором будем рисовать)

vector<string> canvas(CANVAS\_HEIGHT, string(CANVAS\_WIDTH, ' '));

// Функция, которая рисует вершину под номером id

void drawVertex(int id, bool visited = false) {

// Получаем положение вершины по ее номеру из мапы с позициями

pair<int, int> pos = positions[id];

int row = pos.first;

int col = pos.second;

// Выбираем как отобразить вершину, как \* (если посетили)или ее номер (если не посещали)

string label = visited ? "[\*]" : "[" + to\_string(id) + "]";

for (int i = 0; i < 3; ++i) {

if (col + i < (int)canvas[0].size()) {

canvas[row][col + i] = label[i];

}

}

}

// Функция, которая рисует связи между вершинами

void drawEdge(int from, int to) {

// Берем положения вершин

pair<int, int> pos1 = positions[from];

pair<int, int> pos2 = positions[to];

int y1 = pos1.first, x1 = pos1.second;

int y2 = pos2.first, x2 = pos2.second;

if (y1 == y2) {

// Рисуем - для связи вершин у которых одинаковый y

for (int x = min(x1, x2) + 3; x < max(x1, x2); ++x) {

canvas[y1][x] = '-';

}

}

else if (x1 == x2) {

// Рисуем | для связи вершин у которых одинаковый x

for (int y = min(y1, y2) + 1; y < max(y1, y2); ++y) {

canvas[y][x1 + 1] = '|';

}

}

else {

// Диагональ

int dy = (y2 > y1) ? 1 : -1; // тернарный оператор (если y2>y1) то dy = 1, (если y2 <= y1) то dy = -1

int dx = (x2 > x1) ? 1 : -1;

int y = y1 + 1, x = x1 + 2;

// Рисуем диагональные линии

if (from != 4 && to != 4) {

while (y != y2 && x != x2) {

if (y >= 0 && y < CANVAS\_HEIGHT && x >= 0 && x < CANVAS\_WIDTH) {

canvas[y][x] = (dy == dx) ? '\\' : '/';

}

y += dy; // движемся по вертикали к второй вершине

x += dx; // движемся по горизонтали к второй вершине

}

}

}

}

void drawGraph(const set<int>& visited) {

// перед рисованием, очищаем холст от прошлых рисунков

canvas.assign(CANVAS\_HEIGHT, string(CANVAS\_WIDTH, ' '));

// проходим про графу

for (const auto& item : graph) {

// получаем номер каждой вершины

int id = item.first;

// рисуем вершину, и передаем информацию, проходили ли мы через эту вершину

drawVertex(id, visited.count(id) > 0);

}

// проходим про графу

for (const auto& item : graph) {

// получаем номер каждой вершины

int from = item.first;

const vector<int>& toList = item.second;

// от одной вершины может выходить несколько связей, поэтому мы проходим по всем toList (по всем вершинам в которые будет рисоваться связь)

for (int to : toList) {

// у нас неориентированный граф, то есть если есть ребро между 1 и 2, оно записано у обеих вершин:

// graph[1] = {2}

// graph[2] = {1}

// поэтому мы рисуем только тогда, когда from < to

if (from < to) {

drawEdge(from, to);

}

}

}

// отрисовка

for (const string& line : canvas) {

cout << line << '\n';

}

cout << endl;

}

// алгоритм поиска

void dfs(int start) {

set<int> visited; // Множество посещённых вершин, чтобы не заходить в одну и ту же дважды

stack<int> s; // Стек для хранения вершин, которые надо посетить (стек это структура данных, работающая по правилу LIFO)

s.push(start); // Начинаем обход с вершины start — кладём её в стек

while (!s.empty()) {

int current = s.top(); // Пока есть вершины в стеке, берём вершину сверху стека (последнюю добавленную)

s.pop(); // Убираем её из стека, т.к. сейчас будем её обрабатывать

if (visited.count(current))

continue; // Если вершина уже посещена, пропускаем и берём следующую

visited.insert(current); // Отмечаем вершину как посещённую

drawGraph(visited); // Рисуем граф с учётом посещённых вершин (для визуализации процесса)

sleep\_for(milliseconds(3000)); // Пауза, чтобы можно было увидеть текущий шаг

const vector<int>& neighbors = graph[current]; // Получаем соседей текущей вершины

// Проходим соседей в обратном порядке (чтобы обход был в "правильном" порядке)

// rbegin - начало обхода в обратном порядке

for (auto it = neighbors.rbegin(); it != neighbors.rend(); ++it) {

if (!visited.count(\*it)) // Если сосед ещё не посещён

s.push(\*it); // Добавляем его в стек для будущего посещения

}

}

}

int main() {

dfs(5);

return 0;

}