ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы»

Выполнил работу

Апрелков Михаил

Академическая группа J3113

Принято

Лектор, Ходненко Иван Владимирович

Санкт-Петербург

2024

**1. Введение**

# Цель работы

Найти путь алгоритмами\* от места вашего постоянного проживания в спб к университету ИТМО на переданном графе

**Задачи:**

\*Алгоритмы: BFS, DFS, Дейкстра

## **2. Реализация**

**Класс Node**: Представляет узел графа с координатами (x, y). Реализованы перегрузки операторов (<, ==) и хэш-функция для использования в контейнерах.

**Класс Graph**: Представляет граф через список смежности (std::map<Node, std::vector<std::pair<Node, float>>>), где для каждого узла хранится список соседей и вес каждого ребра. (внутри него были реализованы алгоритмы нахождения кратчайшего пути)

Для работы с графом были использованы структуры данных, такие как std::unordered\_set для хранения посещённых узлов, std::unordered\_map для хранения расстояний и предыдущих узлов, а также std::priority\_queue для эффективного выбора узлов с минимальными расстояниями в алгоритме Дейкстры. Каждому узлу в графе соответствует набор рёбер, хранящихся в виде пары (соседний узел, вес рёбер).

Хэш-функция NodeHash для хранения пользовательских объектов Node в контейнерах.

Кроме “дефолтных” библиотек необходимых практически для любой лабораторной работы также использовались:

<map> и <unordered\_map> — для представления графа и хранения предков.

<queue> — для BFS и алгоритма Дейкстры.

<chrono> — для замеров времени выполнения.

<cmath> — для математических операций

<iomanip> — для форматирования вывода.

<limits> — для задания максимальных значений переменных

## **3. Экспериментальная часть**

**Оценка времени выполнения алгоритмов**

|  |  |  |
| --- | --- | --- |
| Алгоритм | Оценка времени работы |  |
| DFS | 5.6165 sec |  |
| BFS | 3.326211 sec |  |
| Dijkstra | 4.261821 sec |  |
|  |  |  |

**Оценка памяти и асимптотики для алгоритмов DFS, BFS, и алгоритма Дейкстры:**

## **1. DFS**

DFS посещает все узлы и все рёбра графа, поэтому

*временная сложность:* ***O(V+E)****.*

(V — количество вершин, E — количество рёбер)

В данном алгоритме хранится множество посещённых узлов: O(V) Максимальная глубина стека вызовов равна числу узлов на самом длинном пути <=O(V).

### 2. BFS

BFS посещает все узлы и все рёбра графа, аналогично DFS,

*временная сложность:* ***O(V+E)***

Хранится очередь узлов для обработки, где в худшем случае одновременно могут находиться все узлы одного уровня. Для полного графа это будет O(V)

Также используется множество посещённых узлов: O(V)

### 3. Алгоритм Дейкстры

Основная операция — извлечение минимального элемента из приоритетной очереди. С использованием кучи

## временная сложность: **O((V+E)**⋅**log V))**

(тк цикл работает за O(V+E), ведь каждое ребро и каждая вершина обрабатываются один раз. А при использовании приоритетной очереди с кучей сложность извлечения и вставки составляет O(log V). Поскольку каждое ребро может потребовать обновления приоритетной очереди, общая сложность такова)

Приоритетная очередь для хранения узлов: O(V).

Хранение расстояний для всех узлов: O(V).

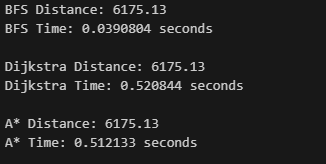
Множество посещённых узлов или аналогичный механизм: O(V).

### 4. Выводы

Данная лабораторная работа оказалось самой временно затратной, однако и очень интересной. Поразмыслив над полученными результатами работы алгоритмов, видно, что для такого относительного большого графа с небольшими степенями вершин bfs показывает себя лучше dfs-a (что в целом было ожидаемо тк bfs это буквально обход в ширину, в своей реализации он быстрее исчерпывает всех возможных соседней вершины текущего уровня и при условии не очень длинных путей между 2мя узлами он определит их всех быстрее), но тк bfs не учитывает вес ребер, то естественно для нахождения кратчайших путей в взвешенном графе намного лучше использовать дейкстру (тк существуют случаи когда проход по 3 ребрам более выгодный нежели по 1, но с большим весом, что собственно дейкстра и умеет замечать, в том числе за счет реализованной приоритетной очереди)

(хотя так как по условиям задачи в городе происходит апокалипсис, то все варианты алгоритмов хороши ).

### 5. Заключение



В ходе выполнения работы были реализованы и протестированы четыре основных алгоритма поиска пути в графе: **BFS (поиск в ширину)**, **DFS (поиск в глубину)**, **Алгоритм Дейкстры**. Каждый из этих алгоритмов имеет свои особенности и область применения.

Таким образом, выбор алгоритма зависит от специфики задачи. Для поиска кратчайшего пути в графах с равными весами рёбер предпочтителен **BFS**. Для задач с графами с разными весами рёбер оптимальна **Дейкстра**.

В целом, каждый алгоритм имеет своё место в различных сценариях использования, и их выбор должен опираться на характер и размер графа, а также на требования к времени выполнения.

# Приложения

#include <iostream>

#include <vector>

#include <string>

#include <sstream>

#include <fstream>

#include <unordered\_map>

#include <unordered\_set>

#include <queue>

#include <cmath>

#include <limits>

#include <ctime>

#include <algorithm>

#include <chrono>

using namespace std;

// Структура узла

struct Node {

    string lon, lat;  // Долгота и широта хранятся как строки

    vector<pair<Node\*, double>> neighbors;  // Список соседей с весами

};

template <typename T, typename... Args>

double measureTime(T func, Args&&... args) {

    auto start = chrono::high\_resolution\_clock::now();

    func(forward<Args>(args)...);  // Execute the function

    auto end = chrono::high\_resolution\_clock::now();

    chrono::duration<double> duration = end - start;

    return duration.count();

}

// Структура графа

struct Graph {

    vector<Node\*> nodes;

    unordered\_map<string, Node\*> nodeMap;  // Хранит узлы по ключу "долгота,широта"

    explicit Graph(const string& filename) {

        loadGraphFromFile(filename);

    }

    // Загрузка графа из файла

    void loadGraphFromFile(const string& filename) {

        ifstream file(filename);

        if (!file.is\_open()) {

            cerr << "Ошибка открытия файла: " << filename << endl;

            return;

        }

        string line;

        while (getline(file, line)) {

            stringstream ss(line);

            string token;

            // Парсинг родительского узла

            getline(ss, token, ':');

            string parentKey = token;  // Ключ - исходная строка координат

            Node\* parentNode;

            if (nodeMap.find(parentKey) == nodeMap.end()) {

                stringstream parentCoordSS(token);

                getline(parentCoordSS, token, ',');

                string parentLon = token;

                getline(parentCoordSS, token, ',');

                string parentLat = token;

                parentNode = new Node{parentLon, parentLat};

                nodes.push\_back(parentNode);

                nodeMap[parentKey] = parentNode;

            } else {

                parentNode = nodeMap[parentKey];

            }

            // Парсинг дочерних узлов и ребер

            while (getline(ss, token, ';')) {

                stringstream childSS(token);

                getline(childSS, token, ',');

                string childLon = token;

                getline(childSS, token, ',');

                string childLat = token;

                string childKey = childLon + "," + childLat; // Ключ - строка координат

                // Убираем пробелы из строки

                childLon.erase(remove(childLon.begin(), childLon.end(), ' '), childLon.end());

                childLat.erase(remove(childLat.begin(), childLat.end(), ' '), childLat.end());

                // Выводим значения для отладки

                // cout << "Parsing lon, lat: " << childLon << ", " << childLat << endl;

                // getline(childSS, token, ','); // Теперь токен это weight         //

                // // Выводим значение для отладки                                  //

                // cout << "Parsing weight: " << token << endl;                     //

                if (token.empty()) {

                    cerr << "Empty string encountered before stod!" << endl;

                    exit(EXIT\_FAILURE); // Завершаем программу при ошибке

                }

                double weight = 0.0;

                try {

                    weight = stod(token);

                    // Если удачно, продолжаем

                } catch (const std::invalid\_argument& e) {

                    cerr << "Invalid argument for stod: '" << token << "' - could not convert to double" << endl;

                    exit(EXIT\_FAILURE); // Завершаем программу при ошибке

                } catch (const std::out\_of\_range& e) {

                    cerr << "Out of range error for stod: '" << token << "'" << endl;

                    exit(EXIT\_FAILURE); // Завершаем программу при ошибке

                }

                Node\* childNode;

                if (nodeMap.find(childKey) == nodeMap.end()) {

                    childNode = new Node{childLon, childLat};

                    nodes.push\_back(childNode);

                    nodeMap[childKey] = childNode;

                } else {

                    childNode = nodeMap[childKey];

                }

                parentNode->neighbors.push\_back({childNode, weight});

            }

        }

        file.close();

    }

    // Поиск ближайшего узла к заданным координатам

    Node\* findClosestNode(double lat, double lon) const {

        Node\* closestNode = nullptr;

        double minDistance = numeric\_limits<double>::infinity();

        for (auto\* node : nodes) {

            double nodeLat = stod(node->lat);

            double nodeLon = stod(node->lon);

            double distance = sqrt(pow(lat - nodeLat, 2) + pow(lon - nodeLon, 2));

            if (distance < minDistance) {

                minDistance = distance;

                closestNode = node;

            }

        }

        return closestNode;

    }

private:

    // Создание или получение узла

    Node\* createOrGetNode(const string& key) {

        if (nodeMap.find(key) == nodeMap.end()) {

            stringstream ss(key);

            string lon, lat;

            getline(ss, lon, ',');

            getline(ss, lat, ',');

            Node\* newNode = new Node{lon, lat};

            nodes.push\_back(newNode);

            nodeMap[key] = newNode;

        }

        return nodeMap[key];

    }

};

// Алгоритм Дейкстры

unordered\_map<Node\*, double> dijkstra(const vector<Node\*>& nodes, Node\* startNode) {

    unordered\_map<Node\*, double> distances;

    priority\_queue<pair<double, Node\*>, vector<pair<double, Node\*>>, greater<>> pq;

    for (auto\* node : nodes) {

        distances[node] = numeric\_limits<double>::infinity();

    }

    distances[startNode] = 0.0;

    pq.push({0.0, startNode});

    while (!pq.empty()) {

        double currentDist = pq.top().first;

        Node\* currentNode = pq.top().second;

        pq.pop();

        if (currentDist > distances[currentNode]) continue;

        for (const auto& [neighbor, weight] : currentNode->neighbors) {

            double newDist = currentDist + weight;

            if (newDist < distances[neighbor]) {

                distances[neighbor] = newDist;

                pq.push({newDist, neighbor});

            }

        }

    }

    return distances;

}

// Эвристика для A\*

double heuristic(Node\* a, Node\* b) {

    double lat1 = stod(a->lat), lon1 = stod(a->lon);

    double lat2 = stod(b->lat), lon2 = stod(b->lon);

    return sqrt(pow(lat1 - lat2, 2) + pow(lon1 - lon2, 2));

}

// Алгоритм A\*

unordered\_map<Node\*, double> aStar(const vector<Node\*>& nodes, Node\* startNode, Node\* endNode) {

    unordered\_map<Node\*, double> distances, fScore;

    priority\_queue<pair<double, Node\*>, vector<pair<double, Node\*>>, greater<>> pq;

    for (auto\* node : nodes) {

        distances[node] = numeric\_limits<double>::infinity();

        fScore[node] = numeric\_limits<double>::infinity();

    }

    distances[startNode] = 0.0;

    fScore[startNode] = heuristic(startNode, endNode);

    pq.push({fScore[startNode], startNode});

    while (!pq.empty()) {

        Node\* current = pq.top().second;

        pq.pop();

        if (current == endNode) break;

        for (const auto& [neighbor, weight] : current->neighbors) {

            double tentativeGScore = distances[current] + weight;

            if (tentativeGScore < distances[neighbor]) {

                distances[neighbor] = tentativeGScore;

                fScore[neighbor] = tentativeGScore + heuristic(neighbor, endNode);

                pq.push({fScore[neighbor], neighbor});

            }

        }

    }

    return distances;

}

// BFS

double bfs(const vector<Node\*>& nodes, Node\* startNode, Node\* endNode) {

    queue<pair<Node\*, double>> q;

    unordered\_set<Node\*> visited;

    q.push({startNode, 0.0});

    visited.insert(startNode);

    while (!q.empty()) {

        Node\* current = q.front().first;

        double currentDist = q.front().second;

        q.pop();

        if (current == endNode) return currentDist;

        for (const auto& [neighbor, weight] : current->neighbors) {

            if (visited.insert(neighbor).second) {

                q.push({neighbor, currentDist + weight});

            }

        }

    }

    return -1.0;

}

// DFS

double dfs(Node\* startNode, Node\* endNode, unordered\_set<Node\*>& visited) {

    if (startNode == endNode) return 0.0;

    visited.insert(startNode);

    for (const auto& [neighbor, weight] : startNode->neighbors) {

        if (visited.find(neighbor) == visited.end()) {

            double result = dfs(neighbor, endNode, visited);

            if (result != -1.0) return result + weight;

        }

    }

    return -1.0;

}

// Точка входа

int main() {

    Graph graph("spb\_graph.txt");

    double myLat = 59.9532059, myLon = 30.2155522;

    double finLat = 59.956248, finLon = 30.309215;

    Node\* startNode = graph.findClosestNode(myLat, myLon);

    Node\* endNode = graph.findClosestNode(finLat, finLon);

    if (!startNode || !endNode) {

        cerr << "Ошибка: начальная или конечная вершина не найдена." << endl;

        return 1;

    }

    unordered\_set<Node\*> visited;

    // cout << "DFS Distance: " << dfs(startNode, endNode, visited) << endl;

    // double dfsTime = measureTime(dfs, startNode, endNode, ref(visited)); // Pass visited by reference

    // cout << "DFS Time: " << dfsTime << " seconds" << endl;

    double bfsTime = measureTime(bfs, graph.nodes, startNode, endNode);

    cout << "BFS Distance: " << bfs(graph.nodes, startNode, endNode) << endl;

    cout << "BFS Time: " << bfsTime << " seconds" << endl;

    double dijkstraTime = measureTime(dijkstra, graph.nodes, startNode);

    auto distancesDijkstra = dijkstra(graph.nodes, startNode);

    cout << endl << "Dijkstra Distance: " << distancesDijkstra[endNode] << endl;

    cout << "Dijkstra Time: " << dijkstraTime << " seconds" << endl;

    double aStarTime = measureTime(aStar, graph.nodes, startNode, endNode);

    auto distancesAStar = aStar(graph.nodes, startNode, endNode);

    cout << endl << "A\* Distance: " << distancesAStar[endNode] << endl;

    cout << "A\* Time: " << aStarTime << " seconds" << endl;

    return 0;

}