ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Апокалипсис - графы»

Выполнил работу

Васькин Виктор

Академическая группа №J3110

Принято

Вершинин Владислав Константинович

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель данной лабораторной работы заключается в изучении алгоритмов поиска кратчайшего пути в графах, а также нахождении кратчайший путь от места проживания до университета ITMO.

В ходе работы проводится анализ и реализация нескольких методов, которые позволяют находить кратчайшие пути, включая алгоритмы поиска в ширину (BFS), поиска в глубину (DFS), алгоритм Дейкстры и алгоритм A\*.

Для достижения главной цели необходимо решить следующие задачи:

1. Ознакомиться с основными принципами работы графов и их представления в программном коде.
2. Реализовать структуру данных для представления графа, включая узлы и рёбра.
3. Реализовать алгоритмы поиска и продемонстрировать их работу с помощью тестовых случаев.
4. Протестировать алгоритмы на различных графах, проанализировать их производительность и сравнить результаты.
5. Применить изученные алгоритмы на реальных данных, считанных из файла, для нахождения кратчайшего пути между двумя заданными точками.
6. Реализация

В процессе выполнения лабораторной работы я следовал нескольким ключевым этапам, каждый из которых направлен на создание алгоритмов для нахождения кратчайших путей в графе.

* 1. Создание структуры данных

Основанная структура графа была определена с помощью классов Node и Graph.

//======================================================================

// 1. Описание структуры Node и класса Graph

//======================================================================

/\*\*

\* Структура Node описывает вершину графа, состоящую из:

\* - координат (lon, lat),

\* - списка рёбер (edges), где каждая пара: (указатель на соседний узел, вес ребра).

\*/

struct Node {

double lon = 0.0; // долгота (например, X-координата)

double lat = 0.0; // широта (например, Y-координата)

// Список рёбер: вектор пар (указатель на соседа, вес).

std::vector<std::pair<Node\*, double>> edges;

};

/\*\*

\* Класс Graph хранит:

\* - множество узлов (Node) в виде unique\_ptr<Node>,

\* - unordered\_map для быстрого доступа к узлу по строковому ключу "lon,lat".

\* Имеет методы для:

\* - добавления узлов,

\* - чтения графа из файла,

\* - поиска ближайшего узла к заданным координатам,

\* - вывода пути и т.д.

\*/

class Graph {

public:

/\*\*

\* Добавить узел с координатами (lon, lat).

\* Если узел с такими координатами уже есть, вернуть существующий.

\* Если нет — создать новый, сохранить и вернуть.

\*/

Node\* addNode(double lon, double lat) {

// Формируем уникальный ключ вида "lon,lat"

std::ostringstream keyStream;

keyStream << std::fixed << std::setprecision(10) << lon << "," << lat;

std::string key = keyStream.str();

auto it = node\_map.find(key);

if (it != node\_map.end()) {

// Узел уже существует, возвращаем его

return it->second;

}

// Иначе создаём новый узел

auto node = std::make\_unique<Node>();

node->lon = lon;

node->lat = lat;

Node\* nodePtr = node.get(); // "сырой" указатель для использования внутри Graph

// Сохраняем узел

nodes.push\_back(std::move(node));

node\_map[key] = nodePtr;

return nodePtr;

}

/\*\*

\* Получить узел по координатам (lon, lat).

\* Если узел не найден, вернуть nullptr.

\*/

Node\* getNode(double lon, double lat) {

std::ostringstream keyStream;

keyStream << std::fixed << std::setprecision(10) << lon << "," << lat;

std::string key = keyStream.str();

auto it = node\_map.find(key);

if (it != node\_map.end()) {

return it->second;

}

return nullptr;

}

/\*\*

\* Считать граф из файла формата:

\* lon1,lat1:lon2,lat2,weight;lon3,lat3,weight;...

\* Каждая строка:

\* "координата родителя:координата ребёнка,вес;координата ребёнка,вес;..."

\*/

void readFromFile(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Не удалось открыть файл: " << filename << "\n";

return;

}

std::string line;

// Считываем файл построчно

while (std::getline(file, line)) {

// Пример строки:

// "lon1,lat1:lon2,lat2,weight;lon3,lat3,weight;..."

std::istringstream lineStream(line);

std::string parentData, edgesData;

// Отделяем часть до двоеточия как "родительские" координаты (lon1,lat1)

if (std::getline(lineStream, parentData, ':')) {

// Заменяем запятые на пробелы, чтобы считать через >>, >> ...

std::replace(parentData.begin(), parentData.end(), ',', ' ');

std::istringstream parentStream(parentData);

double lon1, lat1;

// Пытаемся считать lon1 и lat1

if (!(parentStream >> lon1 >> lat1)) {

std::cerr << "[ERROR] Ошибка парсинга узла: " << parentData << "\n";

continue; // Переходим к следующей строке

}

// Добавляем/получаем указатель на родительский узел

Node\* parentNode = addNode(lon1, lat1);

// После двоеточия идут "ребра", разделённые ';'

while (std::getline(lineStream, edgesData, ';')) {

// Каждое ребро: "lon2,lat2,weight"

std::replace(edgesData.begin(), edgesData.end(), ',', ' ');

std::istringstream edgeStream(edgesData);

double lon2, lat2, weight;

if (!(edgeStream >> lon2 >> lat2 >> weight)) {

std::cerr << "[ERROR] Ошибка парсинга ребра: " << edgesData << "\n";

continue;

}

// Добавляем/получаем указатель на узел-ребёнок

Node\* childNode = addNode(lon2, lat2);

// Добавляем связь (ребро) в обе стороны (неориентированный граф)

parentNode->edges.emplace\_back(childNode, weight);

childNode->edges.emplace\_back(parentNode, weight);

}

}

}

file.close();

}

/\*\*

\* Найти в графе узел, который ближе всех к точке (lat, lon)

\*/

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

double minDist = std::numeric\_limits<double>::max();

Node\* closest = nullptr;

// Простой перебор (можно было бы использовать kd-tree, но для лабы )

for (auto& up : const\_cast<Graph\*>(this)->nodes) {

Node\* n = up.get();

double dx = n->lon - lon;

double dy = n->lat - lat;

double dist = std::sqrt(dx \* dx + dy \* dy);

if (dist < minDist) {

minDist = dist;

closest = n;

}

}

return closest;

}

/\*\*

\* Вывести последовательность узлов (путь) и подсчитать суммарный вес

\* (перебирая рёбра между подряд идущими узлами в path).

\*/

void printPath(const std::vector<Node\*>& path) const {

if (path.empty()) {

std::cout << "Путь не найден.\n";

return;

}

double totalWeight = 0.0;

std::cout << "Путь:\n";

for (size\_t i = 0; i < path.size(); ++i) {

// Печатаем координаты каждого узла

std::cout << "(" << path[i]->lat << ", " << path[i]->lon << ")";

if (i < path.size() - 1) {

std::cout << " -> ";

// Ищем вес ребра между path[i] и path[i+1]

Node\* cur = path[i];

Node\* nxt = path[i + 1];

for (auto& edge : cur->edges) {

if (edge.first == nxt) {

totalWeight += edge.second;

break;

}

}

}

}

std::cout << "\nОбщая длина пути: " << totalWeight << "\n";

}

/\*\*

\* Печать всего графа, для отладки (я так и не понял нужен он или нет):

\* Выводит для каждого узла все рёбра.

\*/

void printGraph() const {

std::cout << "=== Граф ===\n";

for (auto& up : const\_cast<Graph\*>(this)->nodes) {

Node\* n = up.get();

std::cout << "Узел (" << n->lat << ", " << n->lon << "):\n";

for (auto& edge : n->edges) {

std::cout << " -> (" << edge.first->lat << ", "

<< edge.first->lon << "), вес=" << edge.second << "\n";

}

}

}

private:

// Храним все узлы в векторе unique\_ptr, чтобы не заботиться о delete вручную

std::vector<std::unique\_ptr<Node>> nodes;

// Карта ключ -> указатель на узел

// Ключ = "lon,lat" (строка), чтобы не плодить дубли

std::unordered\_map<std::string, Node\*> node\_map;

};

* 1. Реализация алгоритма BFS

/\*\*

\* Алгоритм BFS (поиск в ширину):

\* - Работает по принципу очереди.

\* - Находит кратчайший путь в плане количества рёбер в невзвешенном графе

\* - Не учитывает веса.

\*/

std::vector<Node\*> BFS(Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, bool> visited; // Были ли уже в узле

std::unordered\_map<Node\*, Node\*> cameFrom; // Откуда пришли в этот узел

std::queue<Node\*> q; // Очередь узлов на обработку

q.push(start);

visited[start] = true;

while (!q.empty()) {

Node\* current = q.front();

q.pop();

if (current == goal) {

// Восстанавливаем путь, двигаясь назад по cameFrom

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Идём по всем соседям current

for (auto& edge : current->edges) {

Node\* neighbor = edge.first;

if (!visited[neighbor]) {

visited[neighbor] = true;

cameFrom[neighbor] = current;

q.push(neighbor);

}

}

}

// Путь не найден (((

return {};

}

2.3 Реализация алгоритма DFS

/\*\*

\* Алгоритм DFS (поиск в глубину):

\* - Использует стек.

\* - Не ищет кратчайший путь по весам, а просто идёт «в глубину».

\* - Как только находит goal, возвращает путь.

\*/

std::vector<Node\*> DFS(Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, bool> visited;

std::unordered\_map<Node\*, Node\*> cameFrom;

std::stack<Node\*> st;

st.push(start);

visited[start] = true;

while (!st.empty()) {

Node\* current = st.top();

st.pop();

if (current == goal) {

// Восстанавливаем путь

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Обходим соседей

for (auto& edge : current->edges) {

Node\* neighbor = edge.first;

if (!visited[neighbor]) {

visited[neighbor] = true;

cameFrom[neighbor] = current;

st.push(neighbor);

}

}

}

// Путь не найден

return {};

}

* 1. Реализация алгоритма Дейкстра

/\*\*

\* Алгоритм Дейкстры:

\* - Позволяет находить кратчайший путь в графе с неотрицательными весами рёбер.

\* - Используем приоритетную очередь, где расстояние минимальное.

\*/

std::vector<Node\*> Dijkstra(Graph& graph, Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, double> dist; // dist[x] = расстояние от start до x

std::unordered\_map<Node\*, Node\*> cameFrom; // восстановление пути

std::unordered\_map<Node\*, bool> visited;

dist[start] = 0.0; // Для стартовой вершины расстояние = 0

using PQItem = std::pair<double, Node\*>;

// Очередь с приоритетом: вершина с наименьшим расстоянием на вершине

std::priority\_queue<PQItem, std::vector<PQItem>, std::greater<>> pq;

pq.emplace(0.0, start);

while (!pq.empty()) {

// Извлекаем узел с минимальным расстоянием

PQItem topPair = pq.top();

pq.pop();

double currentDist = topPair.first;

Node\* currentNode = topPair.second;

if (visited[currentNode]) {

continue;

}

visited[currentNode] = true;

// Если дошли до goal, восстанавливаем путь

if (currentNode == goal) {

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// «Релаксация» рёбер:

// если найден путь короче, обновляем dist[neighbor].

for (auto& edge : currentNode->edges) {

Node\* neighbor = edge.first;

double weight = edge.second;

double alt = currentDist + weight;

// Если сосед ещё не встречался, инициализируем inf

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

dist[neighbor] = std::numeric\_limits<double>::infinity();

}

// Если нашли более короткий маршрут к neighbor

if (alt < dist[neighbor]) {

dist[neighbor] = alt;

cameFrom[neighbor] = currentNode;

pq.emplace(alt, neighbor);

}

}

}

// Если очередь опустела, а goal не достигнут, пути нет

return {};

}

* 1. Реализация алгоритма A\*

/\*\*

\* Алгоритм A\*:

\* - Модификация Дейкстры, ускоряется за счёт эвристики (heuristic).

\* - fScore[x] = gScore[x] + h(x), где h(x) — приблизительная дистанция до цели.

\*/

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

double dx = a->lon - b->lon;

double dy = a->lat - b->lat;

// Евклидово расстояние

return std::sqrt(dx \* dx + dy \* dy);

}

std::vector<Node\*> AStar(Graph& graph, Node\* start, Node\* goal) {

if (!start || !goal) return {};

// gScore[x] = расстояние от start до x

// fScore[x] = gScore[x] + эвристика(x, goal)

std::unordered\_map<Node\*, double> gScore;

std::unordered\_map<Node\*, double> fScore;

std::unordered\_map<Node\*, Node\*> cameFrom;

std::unordered\_map<Node\*, bool> visited;

gScore[start] = 0.0;

fScore[start] = heuristic(start, goal);

using PQItem = std::pair<double, Node\*>;

// Очередь с приоритетом по fScore (чем меньше fScore, тем выше приоритет)

std::priority\_queue<PQItem, std::vector<PQItem>, std::greater<>> openSet;

openSet.emplace(fScore[start], start);

while (!openSet.empty()) {

// Извлекаем узел с минимальным fScore

PQItem topPair = openSet.top();

openSet.pop();

double currentF = topPair.first;

Node\* currentNode = topPair.second;

// Если достигли goal, восстанавливаем путь

if (currentNode == goal) {

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Помечаем как посещённый

visited[currentNode] = true;

// Для каждого соседа currentNode

for (auto& edge : currentNode->edges) {

Node\* neighbor = edge.first;

double weight = edge.second;

if (visited[neighbor]) {

// Если уже посетили, не пытаемся улучшить

continue;

}

double tentative\_gScore = gScore[currentNode] + weight;

// Инициализируем inf при первом посещении

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

gScore[neighbor] = std::numeric\_limits<double>::infinity();

}

// Если нашли более короткий путь к neighbor

if (tentative\_gScore < gScore[neighbor]) {

cameFrom[neighbor] = currentNode;

gScore[neighbor] = tentative\_gScore;

// fScore = gScore + эвристика

fScore[neighbor] = tentative\_gScore + heuristic(neighbor, goal);

openSet.emplace(fScore[neighbor], neighbor);

}

}

}

// Путь не найден

return {};

}

2.6 Тестирование

//======================================================================

// 3. Тестовые функции

//======================================================================

/\*\*

\* Вспомогательная функция, чтобы вычислить

\* суммарный вес пути (vector<Node\*>) через последовательный перебор рёбер.

\*/

double computePathWeight(const std::vector<Node\*>& path) {

double total = 0.0;

for (size\_t i = 0; i + 1 < path.size(); i++) {

Node\* cur = path[i];

Node\* nxt = path[i + 1];

// ищем ребро cur->nxt

for (auto& e : cur->edges) {

if (e.first == nxt) {

total += e.second;

break;

}

}

}

return total;

}

/\*\*

\* Тест №1: создаём линейный граф: n1 -> n2 -> n3 -> n4

\* (каждый переход вес = 1), проверяем, что все алгоритмы

\* (BFS, DFS, Dijkstra, A\*) находят путь суммарной длиной 3.

\*/

void test\_simple\_only\_thing\_I\_had\_enough\_imagination\_for() {

Graph g;

// Создаём 4 узла

Node\* n1 = g.addNode(1.0, 1.0);

Node\* n2 = g.addNode(2.0, 2.0);

Node\* n3 = g.addNode(3.0, 3.0);

Node\* n4 = g.addNode(4.0, 4.0);

// Соединяем их попарно

n1->edges.emplace\_back(n2, 1.0);

n2->edges.emplace\_back(n1, 1.0);

n2->edges.emplace\_back(n3, 1.0);

n3->edges.emplace\_back(n2, 1.0);

n3->edges.emplace\_back(n4, 1.0);

n4->edges.emplace\_back(n3, 1.0);

// Запускаем алгоритмы

auto bfsPath = BFS(n1, n4);

auto dfsPath = DFS(n1, n4);

auto dijPath = Dijkstra(g, n1, n4);

auto astPath = AStar(g, n1, n4);

// Проверяем, что пути найдены (не пустые)

assert(!bfsPath.empty());

assert(!dfsPath.empty());

assert(!dijPath.empty());

assert(!astPath.empty());

// Считаем итоговое расстояние

double bfsDist = computePathWeight(bfsPath);

double dfsDist = computePathWeight(dfsPath);

double dijDist = computePathWeight(dijPath);

double astDist = computePathWeight(astPath);

// Ожидаем суммарный вес = 3.0

assert(std::fabs(bfsDist - 3.0) < 1e-9);

assert(std::fabs(dfsDist - 3.0) < 1e-9);

assert(std::fabs(dijDist - 3.0) < 1e-9);

assert(std::fabs(astDist - 3.0) < 1e-9);

std::cout << "[test1\_its\_only\_thing\_I\_had\_enough\_imagination\_for] passed!\n";

}

/\*\*

\* Тест №2: создаём два узла без рёбер,

\* проверяем, что все алгоритмы возвращают пустой путь.

\*/

void test2\_no\_path() {

Graph g;

Node\* n1 = g.addNode(10.0, 10.0);

Node\* n2 = g.addNode(20.0, 20.0);

// Никаких рёбер не добавляем, значит пути нет

// BFS

auto bfsPath = BFS(n1, n2);

assert(bfsPath.empty());

// DFS

auto dfsPath = DFS(n1, n2);

assert(dfsPath.empty());

// Dijkstra

auto dijPath = Dijkstra(g, n1, n2);

assert(dijPath.empty());

// A\*

auto astPath = AStar(g, n1, n2);

assert(astPath.empty());

std::cout << "[test2\_no\_path] passed!\n";

}

/\*\*

\* Запускает все тесты по очереди.

\* Если какой-то testX() упадёт в assert,

\* программа завершится с ошибкой.

\*/

void runAllTests() {

test\_simple\_only\_thing\_I\_had\_enough\_imagination\_for();

test2\_no\_path();

// Можно добавить ещё тестовых сцен...

std::cout << "ЙОУ ВСЕ ТЕСТЫ ПРОШЛИ!!!!!!!!!!\n";

}

1. Экспериментальная часть

В ходе эксперимента были протестированы четыре алгоритма поиска кратчайшего пути: BFS, DFS, Дейкстра и A\*. Для каждого алгоритма было измерено время выполнения, которое представлено ниже:

* BFS (Поиск в ширину): 951 мс
* DFS (Поиск в глубину): 672 мс
* Дейкстра: 398 мс
* A\*: 83 мс

1. Заключение

В ходе выполнения лабораторной работы были реализованы четыре алгоритма поиска кратчайшего пути: BFS, DFS, Дейкстра и A\*.

Тестирование, проведенное на графах с различным количеством узлов и рёбер, подтвердило, что алгоритмы A\* и Дейкстра являются наиболее эффективными для нахождения кратчайшего пути в графах, особенно если рёбра имеют положительные веса. Альтернативные методы, такие как BFS и DFS, показали менее удовлетворительные результаты, повредив производительность и не обеспечив гарантии нахождения кратчайшего пути.

В частности, алгоритм A\* продемонстрировал лучшее время исполнения благодаря эвристике, позволяя значительно сократить время поиска. В то время как Дейкстра также обеспечил хорошие результаты, его производительность уступала A\* из-за отсутствия эвристики.

1. Приложения

ПРИЛОЖЕНИЕ

Листинг кода

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <memory>

#include <algorithm>

#include <queue>

#include <stack>

#include <cmath>

#include <limits>

#include <chrono>

#include <iomanip>

#include <cassert>

//======================================================================

// 1. Описание структуры Node и класса Graph

//======================================================================

/\*\*

\* Структура Node описывает вершину графа, состоящую из:

\* - координат (lon, lat),

\* - списка рёбер (edges), где каждая пара: (указатель на соседний узел, вес ребра).

\*/

struct Node {

double lon = 0.0; // долгота (например, X-координата)

double lat = 0.0; // широта (например, Y-координата)

// Список рёбер: вектор пар (указатель на соседа, вес).

std::vector<std::pair<Node\*, double>> edges;

};

/\*\*

\* Класс Graph хранит:

\* - множество узлов (Node) в виде unique\_ptr<Node>,

\* - unordered\_map для быстрого доступа к узлу по строковому ключу "lon,lat".

\* Имеет методы для:

\* - добавления узлов,

\* - чтения графа из файла,

\* - поиска ближайшего узла к заданным координатам,

\* - вывода пути и т.д.

\*/

class Graph {

public:

/\*\*

\* Добавить узел с координатами (lon, lat).

\* Если узел с такими координатами уже есть, вернуть существующий.

\* Если нет — создать новый, сохранить и вернуть.

\*/

Node\* addNode(double lon, double lat) {

// Формируем уникальный ключ вида "lon,lat"

std::ostringstream keyStream;

keyStream << std::fixed << std::setprecision(10) << lon << "," << lat;

std::string key = keyStream.str();

auto it = node\_map.find(key);

if (it != node\_map.end()) {

// Узел уже существует, возвращаем его

return it->second;

}

// Иначе создаём новый узел

auto node = std::make\_unique<Node>();

node->lon = lon;

node->lat = lat;

Node\* nodePtr = node.get(); // "сырой" указатель для использования внутри Graph

// Сохраняем узел

nodes.push\_back(std::move(node));

node\_map[key] = nodePtr;

return nodePtr;

}

/\*\*

\* Получить узел по координатам (lon, lat).

\* Если узел не найден, вернуть nullptr.

\*/

Node\* getNode(double lon, double lat) {

std::ostringstream keyStream;

keyStream << std::fixed << std::setprecision(10) << lon << "," << lat;

std::string key = keyStream.str();

auto it = node\_map.find(key);

if (it != node\_map.end()) {

return it->second;

}

return nullptr;

}

/\*\*

\* Считать граф из файла формата:

\* lon1,lat1:lon2,lat2,weight;lon3,lat3,weight;...

\* Каждая строка:

\* "координата родителя:координата ребёнка,вес;координата ребёнка,вес;..."

\*/

void readFromFile(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Не удалось открыть файл: " << filename << "\n";

return;

}

std::string line;

// Считываем файл построчно

while (std::getline(file, line)) {

// Пример строки:

// "lon1,lat1:lon2,lat2,weight;lon3,lat3,weight;..."

std::istringstream lineStream(line);

std::string parentData, edgesData;

// Отделяем часть до двоеточия как "родительские" координаты (lon1,lat1)

if (std::getline(lineStream, parentData, ':')) {

// Заменяем запятые на пробелы, чтобы считать через >>, >> ...

std::replace(parentData.begin(), parentData.end(), ',', ' ');

std::istringstream parentStream(parentData);

double lon1, lat1;

// Пытаемся считать lon1 и lat1

if (!(parentStream >> lon1 >> lat1)) {

std::cerr << "[ERROR] Ошибка парсинга узла: " << parentData << "\n";

continue; // Переходим к следующей строке

}

// Добавляем/получаем указатель на родительский узел

Node\* parentNode = addNode(lon1, lat1);

// После двоеточия идут "ребра", разделённые ';'

while (std::getline(lineStream, edgesData, ';')) {

// Каждое ребро: "lon2,lat2,weight"

std::replace(edgesData.begin(), edgesData.end(), ',', ' ');

std::istringstream edgeStream(edgesData);

double lon2, lat2, weight;

if (!(edgeStream >> lon2 >> lat2 >> weight)) {

std::cerr << "[ERROR] Ошибка парсинга ребра: " << edgesData << "\n";

continue;

}

// Добавляем/получаем указатель на узел-ребёнок

Node\* childNode = addNode(lon2, lat2);

// Добавляем связь (ребро) в обе стороны (неориентированный граф)

parentNode->edges.emplace\_back(childNode, weight);

childNode->edges.emplace\_back(parentNode, weight);

}

}

}

file.close();

}

/\*\*

\* Найти в графе узел, который ближе всех к точке (lat, lon)

\*/

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

double minDist = std::numeric\_limits<double>::max();

Node\* closest = nullptr;

// Простой перебор (можно было бы использовать kd-tree, но для лабы )

for (auto& up : const\_cast<Graph\*>(this)->nodes) {

Node\* n = up.get();

double dx = n->lon - lon;

double dy = n->lat - lat;

double dist = std::sqrt(dx \* dx + dy \* dy);

if (dist < minDist) {

minDist = dist;

closest = n;

}

}

return closest;

}

/\*\*

\* Вывести последовательность узлов (путь) и подсчитать суммарный вес

\* (перебирая рёбра между подряд идущими узлами в path).

\*/

void printPath(const std::vector<Node\*>& path) const {

if (path.empty()) {

std::cout << "Путь не найден.\n";

return;

}

double totalWeight = 0.0;

std::cout << "Путь:\n";

for (size\_t i = 0; i < path.size(); ++i) {

// Печатаем координаты каждого узла

std::cout << "(" << path[i]->lat << ", " << path[i]->lon << ")";

if (i < path.size() - 1) {

std::cout << " -> ";

// Ищем вес ребра между path[i] и path[i+1]

Node\* cur = path[i];

Node\* nxt = path[i + 1];

for (auto& edge : cur->edges) {

if (edge.first == nxt) {

totalWeight += edge.second;

break;

}

}

}

}

std::cout << "\nОбщая длина пути: " << totalWeight << "\n";

}

/\*\*

\* Печать всего графа, для отладки (я так и не понял нужен он или нет):

\* Выводит для каждого узла все рёбра.

\*/

void printGraph() const {

std::cout << "=== Граф ===\n";

for (auto& up : const\_cast<Graph\*>(this)->nodes) {

Node\* n = up.get();

std::cout << "Узел (" << n->lat << ", " << n->lon << "):\n";

for (auto& edge : n->edges) {

std::cout << " -> (" << edge.first->lat << ", "

<< edge.first->lon << "), вес=" << edge.second << "\n";

}

}

}

private:

// Храним все узлы в векторе unique\_ptr, чтобы не заботиться о delete вручную

std::vector<std::unique\_ptr<Node>> nodes;

// Карта ключ -> указатель на узел

// Ключ = "lon,lat" (строка), чтобы не плодить дубли

std::unordered\_map<std::string, Node\*> node\_map;

};

//======================================================================

// 2. Реализация алгоритмов: BFS, DFS, Dijkstra, A\*

//======================================================================

/\*\*

\* Алгоритм BFS (поиск в ширину):

\* - Работает по принципу очереди.

\* - Находит кратчайший путь в плане количества рёбер в невзвешенном графе

\* - Не учитывает веса.

\*/

std::vector<Node\*> BFS(Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, bool> visited; // Были ли уже в узле

std::unordered\_map<Node\*, Node\*> cameFrom; // Откуда пришли в этот узел

std::queue<Node\*> q; // Очередь узлов на обработку

q.push(start);

visited[start] = true;

while (!q.empty()) {

Node\* current = q.front();

q.pop();

if (current == goal) {

// Восстанавливаем путь, двигаясь назад по cameFrom

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Идём по всем соседям current

for (auto& edge : current->edges) {

Node\* neighbor = edge.first;

if (!visited[neighbor]) {

visited[neighbor] = true;

cameFrom[neighbor] = current;

q.push(neighbor);

}

}

}

// Путь не найден (((

return {};

}

/\*\*

\* Алгоритм DFS (поиск в глубину):

\* - Использует стек.

\* - Не ищет кратчайший путь по весам, а просто идёт «в глубину».

\* - Как только находит goal, возвращает путь.

\*/

std::vector<Node\*> DFS(Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, bool> visited;

std::unordered\_map<Node\*, Node\*> cameFrom;

std::stack<Node\*> st;

st.push(start);

visited[start] = true;

while (!st.empty()) {

Node\* current = st.top();

st.pop();

if (current == goal) {

// Восстанавливаем путь

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Обходим соседей

for (auto& edge : current->edges) {

Node\* neighbor = edge.first;

if (!visited[neighbor]) {

visited[neighbor] = true;

cameFrom[neighbor] = current;

st.push(neighbor);

}

}

}

// Путь не найден

return {};

}

/\*\*

\* Алгоритм Дейкстры:

\* - Позволяет находить кратчайший путь в графе с неотрицательными весами рёбер.

\* - Используем приоритетную очередь, где расстояние минимальное.

\*/

std::vector<Node\*> Dijkstra(Graph& graph, Node\* start, Node\* goal) {

if (!start || !goal) return {};

std::unordered\_map<Node\*, double> dist; // dist[x] = расстояние от start до x

std::unordered\_map<Node\*, Node\*> cameFrom; // восстановление пути

std::unordered\_map<Node\*, bool> visited;

dist[start] = 0.0; // Для стартовой вершины расстояние = 0

using PQItem = std::pair<double, Node\*>;

// Очередь с приоритетом: вершина с наименьшим расстоянием на вершине

std::priority\_queue<PQItem, std::vector<PQItem>, std::greater<>> pq;

pq.emplace(0.0, start);

while (!pq.empty()) {

// Извлекаем узел с минимальным расстоянием

PQItem topPair = pq.top();

pq.pop();

double currentDist = topPair.first;

Node\* currentNode = topPair.second;

if (visited[currentNode]) {

continue;

}

visited[currentNode] = true;

// Если дошли до goal, восстанавливаем путь

if (currentNode == goal) {

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// «Релаксация» рёбер:

// если найден путь короче, обновляем dist[neighbor].

for (auto& edge : currentNode->edges) {

Node\* neighbor = edge.first;

double weight = edge.second;

double alt = currentDist + weight;

// Если сосед ещё не встречался, инициализируем inf

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

dist[neighbor] = std::numeric\_limits<double>::infinity();

}

// Если нашли более короткий маршрут к neighbor

if (alt < dist[neighbor]) {

dist[neighbor] = alt;

cameFrom[neighbor] = currentNode;

pq.emplace(alt, neighbor);

}

}

}

// Если очередь опустела, а goal не достигнут, пути нет

return {};

}

/\*\*

\* Алгоритм A\*:

\* - Модификация Дейкстры, ускоряется за счёт эвристики (heuristic).

\* - fScore[x] = gScore[x] + h(x), где h(x) — приблизительная дистанция до цели.

\*/

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

double dx = a->lon - b->lon;

double dy = a->lat - b->lat;

// Евклидово расстояние

return std::sqrt(dx \* dx + dy \* dy);

}

std::vector<Node\*> AStar(Graph& graph, Node\* start, Node\* goal) {

if (!start || !goal) return {};

// gScore[x] = расстояние от start до x

// fScore[x] = gScore[x] + эвристика(x, goal)

std::unordered\_map<Node\*, double> gScore;

std::unordered\_map<Node\*, double> fScore;

std::unordered\_map<Node\*, Node\*> cameFrom;

std::unordered\_map<Node\*, bool> visited;

gScore[start] = 0.0;

fScore[start] = heuristic(start, goal);

using PQItem = std::pair<double, Node\*>;

// Очередь с приоритетом по fScore (чем меньше fScore, тем выше приоритет)

std::priority\_queue<PQItem, std::vector<PQItem>, std::greater<>> openSet;

openSet.emplace(fScore[start], start);

while (!openSet.empty()) {

// Извлекаем узел с минимальным fScore

PQItem topPair = openSet.top();

openSet.pop();

double currentF = topPair.first;

Node\* currentNode = topPair.second;

// Если достигли goal, восстанавливаем путь

if (currentNode == goal) {

std::vector<Node\*> path;

for (Node\* p = goal; p; p = cameFrom[p]) {

path.push\_back(p);

}

std::reverse(path.begin(), path.end());

return path;

}

// Помечаем как посещённый

visited[currentNode] = true;

// Для каждого соседа currentNode

for (auto& edge : currentNode->edges) {

Node\* neighbor = edge.first;

double weight = edge.second;

if (visited[neighbor]) {

// Если уже посетили, не пытаемся улучшить

continue;

}

double tentative\_gScore = gScore[currentNode] + weight;

// Инициализируем inf при первом посещении

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

gScore[neighbor] = std::numeric\_limits<double>::infinity();

}

// Если нашли более короткий путь к neighbor

if (tentative\_gScore < gScore[neighbor]) {

cameFrom[neighbor] = currentNode;

gScore[neighbor] = tentative\_gScore;

// fScore = gScore + эвристика

fScore[neighbor] = tentative\_gScore + heuristic(neighbor, goal);

openSet.emplace(fScore[neighbor], neighbor);

}

}

}

// Путь не найден

return {};

}

//======================================================================

// 3. Тестовые функции

//======================================================================

/\*\*

\* Вспомогательная функция, чтобы вычислить

\* суммарный вес пути (vector<Node\*>) через последовательный перебор рёбер.

\*/

double computePathWeight(const std::vector<Node\*>& path) {

double total = 0.0;

for (size\_t i = 0; i + 1 < path.size(); i++) {

Node\* cur = path[i];

Node\* nxt = path[i + 1];

// ищем ребро cur->nxt

for (auto& e : cur->edges) {

if (e.first == nxt) {

total += e.second;

break;

}

}

}

return total;

}

/\*\*

\* Тест №1: создаём линейный граф: n1 -> n2 -> n3 -> n4

\* (каждый переход вес = 1), проверяем, что все алгоритмы

\* (BFS, DFS, Dijkstra, A\*) находят путь суммарной длиной 3.

\*/

void test\_simple\_only\_thing\_I\_had\_enough\_imagination\_for() {

Graph g;

// Создаём 4 узла

Node\* n1 = g.addNode(1.0, 1.0);

Node\* n2 = g.addNode(2.0, 2.0);

Node\* n3 = g.addNode(3.0, 3.0);

Node\* n4 = g.addNode(4.0, 4.0);

// Соединяем их попарно

n1->edges.emplace\_back(n2, 1.0);

n2->edges.emplace\_back(n1, 1.0);

n2->edges.emplace\_back(n3, 1.0);

n3->edges.emplace\_back(n2, 1.0);

n3->edges.emplace\_back(n4, 1.0);

n4->edges.emplace\_back(n3, 1.0);

// Запускаем алгоритмы

auto bfsPath = BFS(n1, n4);

auto dfsPath = DFS(n1, n4);

auto dijPath = Dijkstra(g, n1, n4);

auto astPath = AStar(g, n1, n4);

// Проверяем, что пути найдены (не пустые)

assert(!bfsPath.empty());

assert(!dfsPath.empty());

assert(!dijPath.empty());

assert(!astPath.empty());

// Считаем итоговое расстояние

double bfsDist = computePathWeight(bfsPath);

double dfsDist = computePathWeight(dfsPath);

double dijDist = computePathWeight(dijPath);

double astDist = computePathWeight(astPath);

// Ожидаем суммарный вес = 3.0

assert(std::fabs(bfsDist - 3.0) < 1e-9);

assert(std::fabs(dfsDist - 3.0) < 1e-9);

assert(std::fabs(dijDist - 3.0) < 1e-9);

assert(std::fabs(astDist - 3.0) < 1e-9);

std::cout << "[test1\_its\_only\_thing\_I\_had\_enough\_imagination\_for] passed!\n";

}

/\*\*

\* Тест №2: создаём два узла без рёбер,

\* проверяем, что все алгоритмы возвращают пустой путь.

\*/

void test2\_no\_path() {

Graph g;

Node\* n1 = g.addNode(10.0, 10.0);

Node\* n2 = g.addNode(20.0, 20.0);

// Никаких рёбер не добавляем, значит пути нет

// BFS

auto bfsPath = BFS(n1, n2);

assert(bfsPath.empty());

// DFS

auto dfsPath = DFS(n1, n2);

assert(dfsPath.empty());

// Dijkstra

auto dijPath = Dijkstra(g, n1, n2);

assert(dijPath.empty());

// A\*

auto astPath = AStar(g, n1, n2);

assert(astPath.empty());

std::cout << "[test2\_no\_path] passed!\n";

}

/\*\*

\* Запускает все тесты по очереди.

\* Если какой-то testX() упадёт в assert,

\* программа завершится с ошибкой.

\*/

void runAllTests() {

test\_simple\_only\_thing\_I\_had\_enough\_imagination\_for();

test2\_no\_path();

// Можно добавить ещё тестовых сцен...

std::cout << "ЙОУ ВСЕ ТЕСТЫ ПРОШЛИ!!!!!!!!!!\n";

}

//======================================================================

// 4. main()

//======================================================================

int main() {й

setlocale(LC\_ALL, "Russian");

// Сначала прогоняем тесты

runAllTests();

// Потом уже запускаем код с реальными данными

Graph graph;

graph.readFromFile("spb\_graph.txt");

double startLat = 60.028560;

double startLon = 30.403240;

double endLat = 59.957238;

double endLon = 30.308108;

// Ищем ближайшие узлы в графе к заданным координатам

Node\* startNode = graph.findClosestNode(startLat, startLon);

Node\* endNode = graph.findClosestNode(endLat, endLon);

// Проверяем, удалось ли найти узлы

if (!startNode || !endNode) {

std::cerr << "Не найдены узлы для старта/цели.\n";

return 1;

}

// --- BFS ---

{

auto t1 = std::chrono::high\_resolution\_clock::now();

std::vector<Node\*> pathBFS = BFS(startNode, endNode);

auto t2 = std::chrono::high\_resolution\_clock::now();

std::cout << "\n[BFS] Результат:\n";

graph.printPath(pathBFS);

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(t2 - t1).count();

std::cout << "BFS занял " << duration << " мс\n";

}

// --- DFS ---

{

auto t1 = std::chrono::high\_resolution\_clock::now();

std::vector<Node\*> pathDFS = DFS(startNode, endNode);

auto t2 = std::chrono::high\_resolution\_clock::now();

std::cout << "\n[DFS] Результат:\n";

graph.printPath(pathDFS);

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(t2 - t1).count();

std::cout << "DFS занял " << duration << " мс\n";

}

// --- Dijkstra ---

{

auto t1 = std::chrono::high\_resolution\_clock::now();

std::vector<Node\*> pathDij = Dijkstra(graph, startNode, endNode);

auto t2 = std::chrono::high\_resolution\_clock::now();

std::cout << "\n[Dijkstra] Результат:\n";

graph.printPath(pathDij);

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(t2 - t1).count();

std::cout << "Dijkstra занял " << duration << " мс\n";

}

// --- A\* ---

{

auto t1 = std::chrono::high\_resolution\_clock::now();

std::vector<Node\*> pathAStar = AStar(graph, startNode, endNode);

auto t2 = std::chrono::high\_resolution\_clock::now();

std::cout << "\n[A\*] Результат:\n";

graph.printPath(pathAStar);

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(t2 - t1).count();

std::cout << "A\* занял " << duration << " мс\n";

}

return 0;

}