ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы»

Выполнил работу

Игнатенок Филипп

Академическая группа №J3111

Принято

Ментор, Владислав Вершинин

Санкт-Петербург

2024

1. Введение

Цель: реализовать DFS, BFS, алгоритм Дейкстры и A\* на графе города.

Задачи:

- Реализовать работу с графом из текстового файла

* Реализовать структуру вершины (Node) и графа (Graph)
* Написать Алгоритм поиска в глубину (далее DFS)
* Написать Алгоритм поиска в ширину (далее BFS)
* Написать Алгоритм Дейкстры
* Написать A\*
* Расчитать время работы алгоритмов
* Сравнить методы

1. Теоретическая подготовка

Типы данных:

* int - используется для целочисленных значений (например, счётчиков или индексов).
* double - вещественное число для координат узлов, расстояний и весов рёбер.
* string - строковый тип для хранения текстовых данных, таких как ключи узлов и строки из файла.
* vector - динамический массив, используемый для хранения узлов, рёбер и путей.
* pair - пара значений, например, указатель на узел и вес ребра.
* unordered\_map - ассоциативный контейнер для отображения ключей на значения, используется для быстрого поиска.
* queue - очередь, используется для BFS.
* stack - стек, используется для DFS.
* priority\_queue - очередь с приоритетом, используется в алгоритмах Дейкстры и A\*.
* ostringstream - строковый поток для форматирования строк.
* istringstream - строковый поток для разбора текста.
* ifstream - поток для чтения данных из файла.
* chrono::high\_resolution\_clock - тип для получения точного времени.
* chrono::milliseconds - тип для измерения времени в миллисекундах.
* size\_t - целое беззнаковое число для хранения размеров контейнеров или индексов.
* bool - логический тип для представления флагов (например, посещён ли узел).

Алгоритмы:

* BFS

Поиск в ширину (BFS) представляет собой метод линейного обхода графа, который последовательно обрабатывает узлы уровня за уровнем, начиная с исходного узла. Этот алгоритм гарантирует нахождение кратчайшего пути в невзвешенном графе, поскольку он исследует все узлы на текущем расстоянии перед переходом к более удалённым.

* DFS

Поиск в глубину (DFS) работает по другому принципу: он углубляется в граф как можно дальше вдоль каждого пути, прежде чем возвращаться назад. Он часто используется для исследования всех возможных путей или задач, где порядок обхода важен, но не гарантирует нахождение кратчайшего пути.

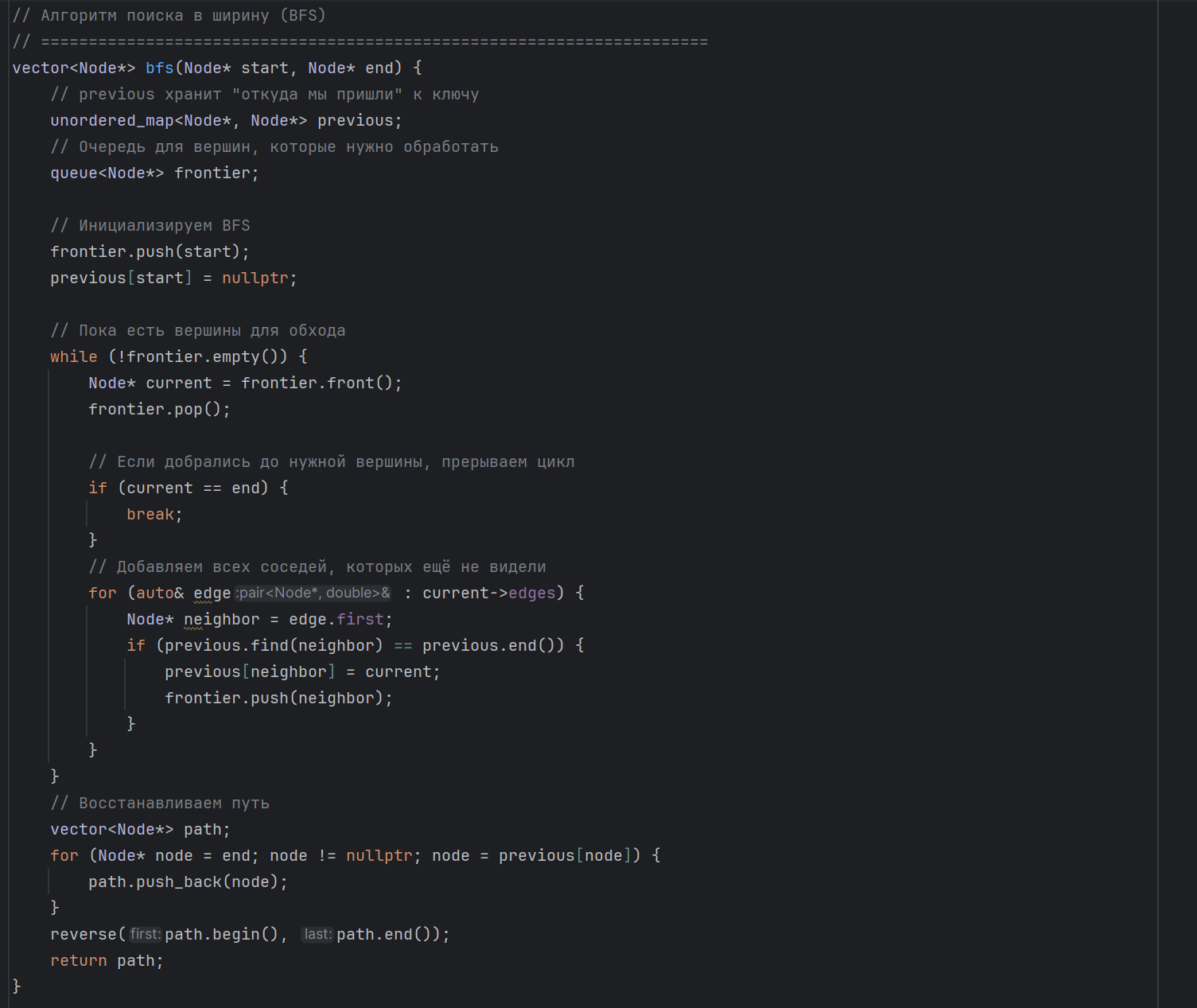
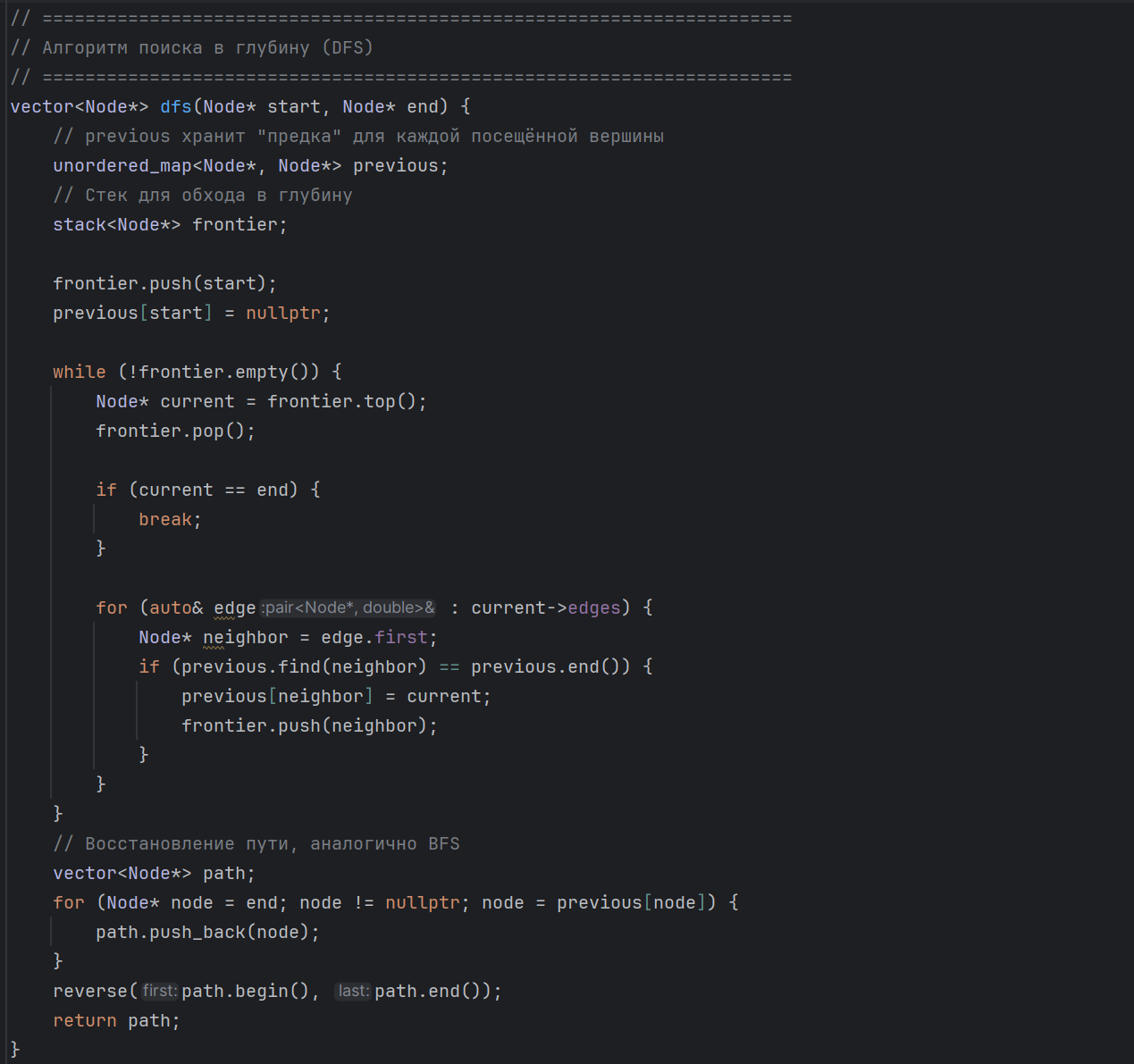
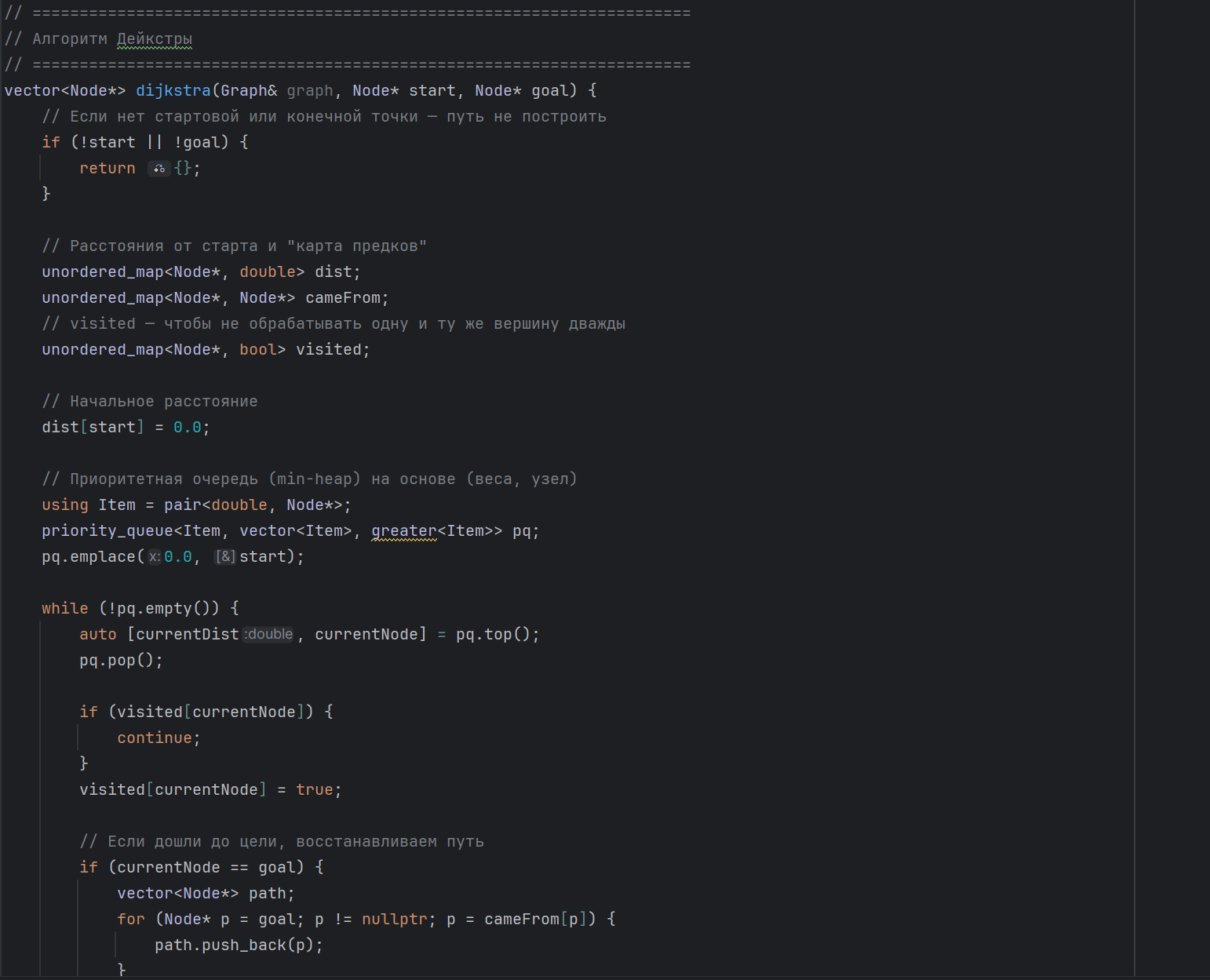
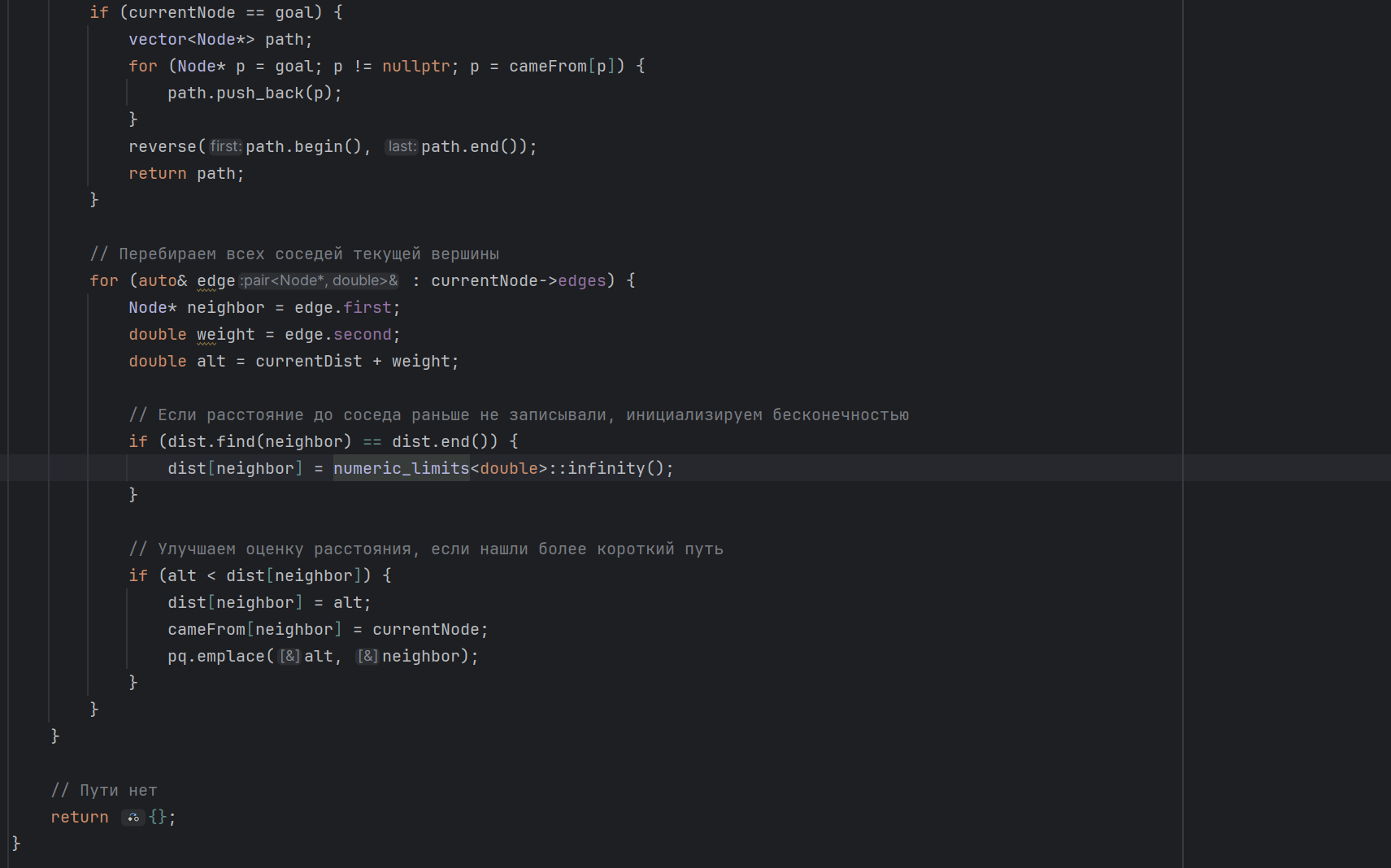
* Дейкстра

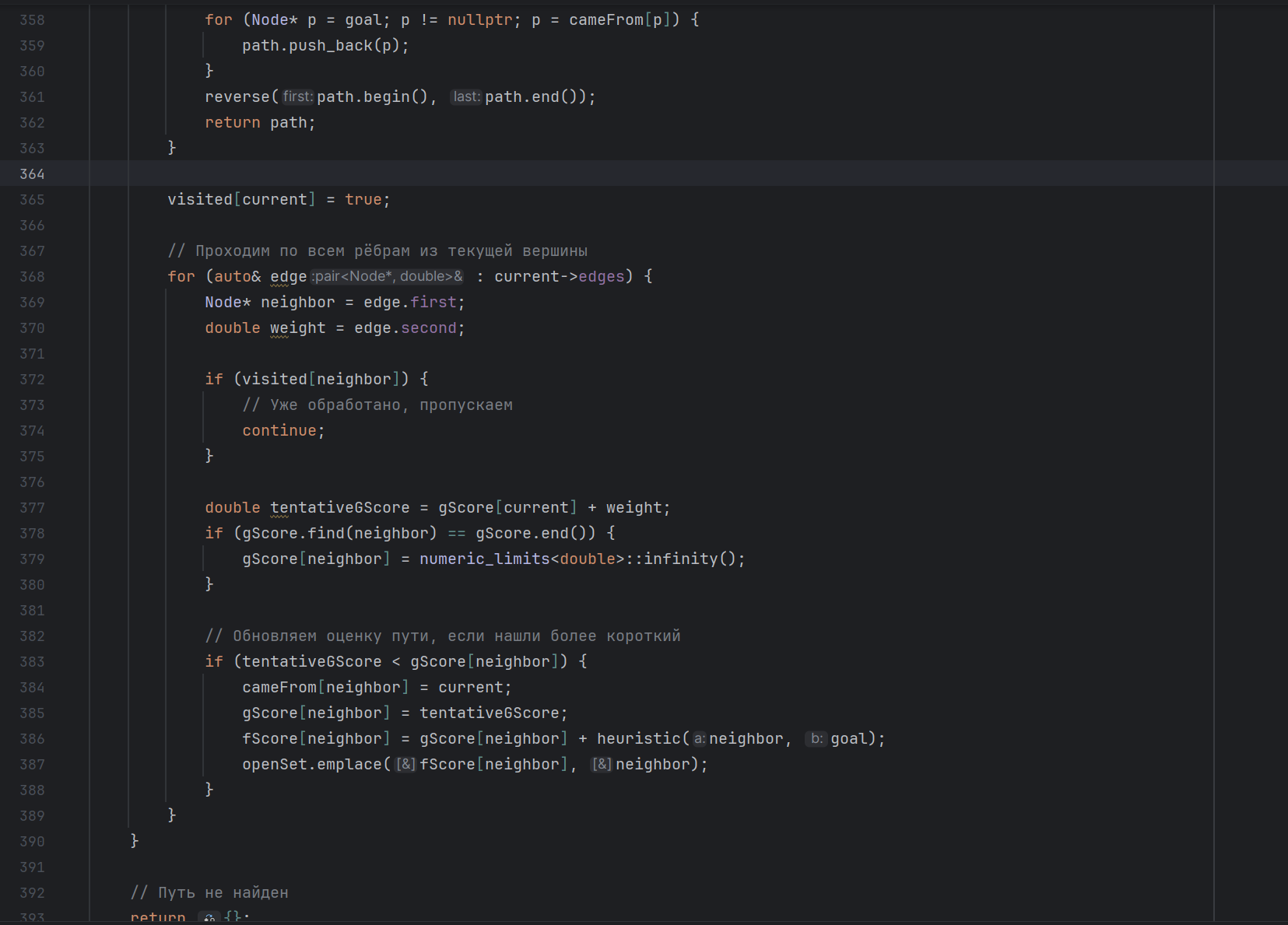
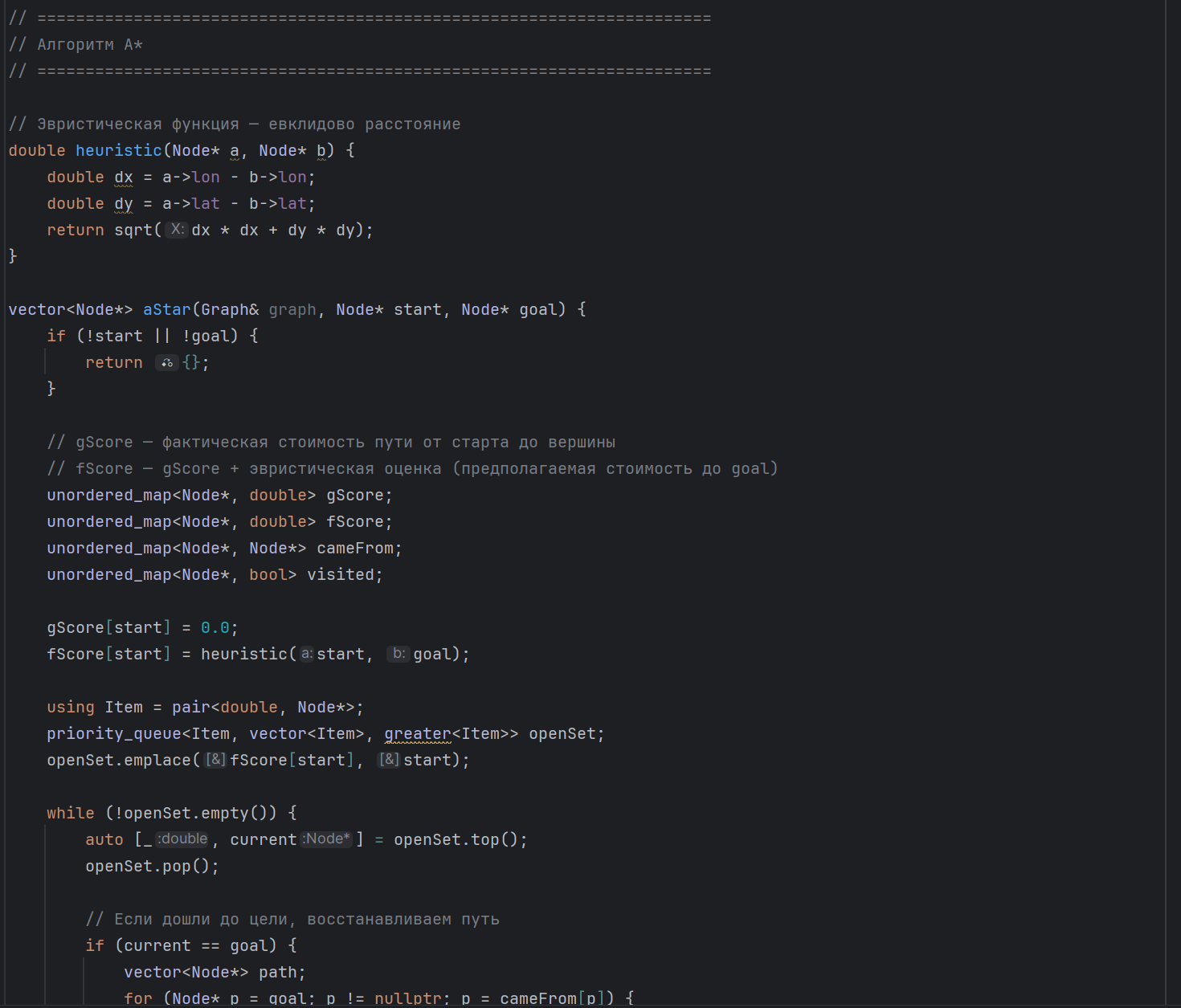
Алгоритм Дейкстры предназначен для нахождения кратчайшего пути в графах с положительными весами рёбер. Он последовательно выбирает вершину с минимальной текущей стоимостью пути из приоритетной очереди и обновляет оценки соседних узлов. Этот метод гарантирует оптимальное решение для графов без отрицательных рёбер.

* A\*

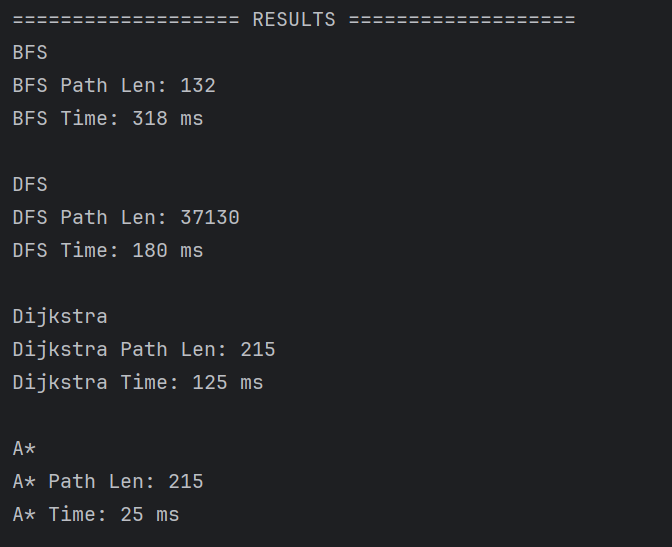
Алгоритм A\* является модификацией Дейкстры, в котором добавлена эвристическая оценка расстояния до конечной вершины. Это позволяет быстрее находить путь, так как узлы приоритизируются не только по текущей стоимости пути, но и по предполагаемой оставшейся дистанции. Эвристика, например евклидово расстояние, делает A\* особенно эффективным для задач, где расстояние до цели может быть точно оценено.

1. Реализация

* BFS
* DFS
* Дейкстра



* A\*

1. Результаты и время работы

Выводы из результатов:

1. A\* работает быстрее Дейкстры
2. A\* находит верное расстояние
3. Дейкстра работает быстрее DFS и DFS
4. Все алгоритмы прошли тестирование
5. Заключение

В ходе работы были реализованы DFS, BFS, алгоритм Дейкстры и A\*.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab8.cpp

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <memory>

#include <unordered\_map>

#include <string>

#include <algorithm>

#include <cmath>

#include <limits>

#include <queue>

#include <utility>

#include <stack>

#include <unordered\_set>

#include <iomanip>

#include <chrono>

#include <cassert>

using namespace std;

using namespace chrono;

struct Node {

double lon = 0.0; // X

double lat = 0.0; // Y

vector<pair<Node\*, double>> edges; // Связи

};

class Graph {

public:

// Добавляем (или возвращаем уже существующий) узел в граф

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

// Формируем строковый ключ с фиксированной точностью

ostringstream buf;

buf << fixed << setprecision(10) << lon << "," << lat;

const string key = buf.str();

// Ищем узел в node\_map

auto found = nodeMap.find(key);

if (found != nodeMap.end()) {

// Узел уже существует - просто возвращаем его

return found->second;

}

// Создаем новый узел и заполняем координаты

unique\_ptr<Node> newNode = make\_unique<Node>();

newNode->lon = lon;

newNode->lat = lat;

Node\* rawPtr = newNode.get();

// Сохраняем узел в векторе и карте

nodes.push\_back(move(newNode));

nodeMap[key] = rawPtr;

return rawPtr;

}

// Получаем указатель на узел по координатам (или nullptr, если узел не найден)

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

ostringstream buf;

buf << fixed << setprecision(10) << lon << "," << lat;

const string key = buf.str();

auto found = nodeMap.find(key);

if (found != nodeMap.end()) {

return found->second;

}

return nullptr;

}

// Считываем граф из текстового файла

void readFromFile(const string& filename) {

ifstream file(filename);

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

cerr << "Failed to open file: " << filename << endl;

return;

}

string line;

while (getline(file, line)) {

// Разделяем строку на часть до двоеточия и после

istringstream lineStream(line);

string parentCoords;

if (!getline(lineStream, parentCoords, ':')) {

continue; // Если не удалось считать "родителя", пропускаем

}

// Заменяем все запятые на пробелы, чтобы проще было парсить

replace(parentCoords.begin(), parentCoords.end(), ',', ' ');

istringstream parentStream(parentCoords);

double lonParent, latParent;

if (!(parentStream >> lonParent >> latParent)) {

cerr << "Error parsing parent node: " << parentCoords << endl;

continue;

}

// Добавляем (или берём существующий) родительский узел

Node\* parentNode = addNode(lonParent, latParent);

// Теперь парсим рёбра, которые перечислены через ';'

string edgePart;

while (getline(lineStream, edgePart, ';')) {

if (edgePart.empty()) {

continue;

}

replace(edgePart.begin(), edgePart.end(), ',', ' ');

istringstream edgeStream(edgePart);

double lonChild, latChild, weight;

if (!(edgeStream >> lonChild >> latChild >> weight)) {

cerr << "Error parsing edge: " << edgePart << endl;

continue;

}

Node\* childNode = addNode(lonChild, latChild);

// Добавляем ребро в обе стороны

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

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

}

}

file.close();

}

// Возвращает ближайший узел к заданным координатам

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

Node\* nearestNode = nullptr;

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

for (auto& nodePtr : nodes) {

Node\* n = nodePtr.get();

double dx = n->lon - lon;

double dy = n->lat - lat;

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

if (dist < minDist) {

minDist = dist;

nearestNode = n;

}

}

return nearestNode;

}

// Печатает путь (список узлов) и суммарную длину

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

if (path.empty()) {

cout << "Path not found." << endl;

return;

}

double totalWeight = 0.0;

cout << "Path:" << endl;

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

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

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

cout << " -> ";

// Взвешенная связь 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;

}

}

}

}

cout << "\nTotal path length: " << totalWeight << endl;

}

// Возвращает все узлы

const vector<unique\_ptr<Node>>& getNodes() const {

return nodes;

}

private:

// Храним все узлы в векторе (для управляемой памяти)

vector<unique\_ptr<Node>> nodes;

// Отображаем строковый ключ "lon,lat" в указатель на соответствующий узел

unordered\_map<string, Node\*> nodeMap;

};

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

// Алгоритм поиска в ширину (BFS)

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

vector<Node\*> bfs(Node\* start, Node\* end) {

// previous хранит "откуда мы пришли" к ключу

unordered\_map<Node\*, Node\*> previous;

// Очередь для вершин, которые нужно обработать

queue<Node\*> frontier;

// Инициализируем BFS

frontier.push(start);

previous[start] = nullptr;

// Пока есть вершины для обхода

while (!frontier.empty()) {

Node\* current = frontier.front();

frontier.pop();

// Если добрались до нужной вершины, прерываем цикл

if (current == end) {

break;

}

// Добавляем всех соседей, которых ещё не видели

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

Node\* neighbor = edge.first;

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

previous[neighbor] = current;

frontier.push(neighbor);

}

}

}

// Восстанавливаем путь

vector<Node\*> path;

for (Node\* node = end; node != nullptr; node = previous[node]) {

path.push\_back(node);

}

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

return path;

}

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

// Алгоритм поиска в глубину (DFS)

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

vector<Node\*> dfs(Node\* start, Node\* end) {

// previous хранит "предка" для каждой посещённой вершины

unordered\_map<Node\*, Node\*> previous;

// Стек для обхода в глубину

stack<Node\*> frontier;

frontier.push(start);

previous[start] = nullptr;

while (!frontier.empty()) {

Node\* current = frontier.top();

frontier.pop();

if (current == end) {

break;

}

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

Node\* neighbor = edge.first;

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

previous[neighbor] = current;

frontier.push(neighbor);

}

}

}

// Восстановление пути, аналогично BFS

vector<Node\*> path;

for (Node\* node = end; node != nullptr; node = previous[node]) {

path.push\_back(node);

}

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

return path;

}

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

// Алгоритм Дейкстры

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

vector<Node\*> dijkstra(Graph& graph, Node\* start, Node\* goal) {

// Если нет стартовой или конечной точки — путь не построить

if (!start || !goal) {

return {};

}

// Расстояния от старта и "карта предков"

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

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

// visited — чтобы не обрабатывать одну и ту же вершину дважды

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

// Начальное расстояние

dist[start] = 0.0;

// Приоритетная очередь (min-heap) на основе (веса, узел)

using Item = pair<double, Node\*>;

priority\_queue<Item, vector<Item>, greater<Item>> pq;

pq.emplace(0.0, start);

while (!pq.empty()) {

auto [currentDist, currentNode] = pq.top();

pq.pop();

if (visited[currentNode]) {

continue;

}

visited[currentNode] = true;

// Если дошли до цели, восстанавливаем путь

if (currentNode == goal) {

vector<Node\*> path;

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

path.push\_back(p);

}

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

return path;

}

// Перебираем всех соседей текущей вершины

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

Node\* neighbor = edge.first;

double weight = edge.second;

double alt = currentDist + weight;

// Если расстояние до соседа раньше не записывали, инициализируем бесконечностью

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

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

}

// Улучшаем оценку расстояния, если нашли более короткий путь

if (alt < dist[neighbor]) {

dist[neighbor] = alt;

cameFrom[neighbor] = currentNode;

pq.emplace(alt, neighbor);

}

}

}

// Пути нет

return {};

}

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

// Алгоритм A\*

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

// Эвристическая функция — евклидово расстояние

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

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

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

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

}

vector<Node\*> aStar(Graph& graph, Node\* start, Node\* goal) {

if (!start || !goal) {

return {};

}

// gScore — фактическая стоимость пути от старта до вершины

// fScore — gScore + эвристическая оценка (предполагаемая стоимость до goal)

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

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

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

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

gScore[start] = 0.0;

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

using Item = pair<double, Node\*>;

priority\_queue<Item, vector<Item>, greater<Item>> openSet;

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

while (!openSet.empty()) {

auto [\_, current] = openSet.top();

openSet.pop();

// Если дошли до цели, восстанавливаем путь

if (current == goal) {

vector<Node\*> path;

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

path.push\_back(p);

}

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

return path;

}

visited[current] = true;

// Проходим по всем рёбрам из текущей вершины

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

Node\* neighbor = edge.first;

double weight = edge.second;

if (visited[neighbor]) {

// Уже обработано, пропускаем

continue;

}

double tentativeGScore = gScore[current] + weight;

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

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

}

// Обновляем оценку пути, если нашли более короткий

if (tentativeGScore < gScore[neighbor]) {

cameFrom[neighbor] = current;

gScore[neighbor] = tentativeGScore;

fScore[neighbor] = gScore[neighbor] + heuristic(neighbor, goal);

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

}

}

}

// Путь не найден

return {};

}

Graph createDemoGraph() {

Graph demoGraph;

// Несколько узлов

Node\* firstNode = demoGraph.addNode(10.0, 20.0);

Node\* secondNode = demoGraph.addNode(15.0, 25.0);

Node\* thirdNode = demoGraph.addNode(20.0, 30.0);

Node\* fourthNode = demoGraph.addNode(25.0, 35.0);

// Связи между узлами

firstNode->edges.emplace\_back(secondNode, 5.0);

secondNode->edges.emplace\_back(thirdNode, 10.0);

thirdNode->edges.emplace\_back(fourthNode, 15.0);

firstNode->edges.emplace\_back(thirdNode, 12.0);

return demoGraph;

}

// Тесты

void testAddNodeFunction() {

Graph testGraph = createDemoGraph();

Node\* n1 = testGraph.getNode(10.0, 20.0);

Node\* n2 = testGraph.getNode(15.0, 25.0);

assert(n1 != nullptr && "Node (10.0, 20.0) не найден!");

assert(n2 != nullptr && "Node (15.0, 25.0) не найден!");

assert(n1->lon == 10.0 && "Долгота узла (10.0, 20.0) некорректна!");

assert(n2->lat == 25.0 && "Широта узла (15.0, 25.0) некорректна!");

cout << "testAddNodeFunction passed!\n";

}

void testGetNodeFunction() {

Graph testGraph = createDemoGraph();

Node\* found = testGraph.getNode(10.0, 20.0);

assert(found != nullptr && "Существующий узел не найден!");

assert(found->lon == 10.0 && "Долгота для (10.0, 20.0) не совпадает!");

assert(found->lat == 20.0 && "Широта для (10.0, 20.0) не совпадает!");

Node\* notFound = testGraph.getNode(30.0, 40.0);

assert(notFound == nullptr && "Несуществующий узел найден!");

cout << "testGetNodeFunction passed!\n";

}

void testBFSAlgorithm() {

Graph testGraph = createDemoGraph();

Node\* startNode = testGraph.getNode(10.0, 20.0);

Node\* finishNode = testGraph.getNode(25.0, 35.0);

vector<Node\*> bfsPath = bfs(startNode, finishNode);

assert(bfsPath.size() >= 2 && "Путь BFS содержит менее двух узлов!");

assert(bfsPath.front() == startNode && "Первый узел пути BFS неправильный!");

assert(bfsPath.back() == finishNode && "Последний узел пути BFS неправильный!");

cout << "testBFSAlgorithm passed!\n";

}

void testDFSAlgorithm() {

Graph testGraph = createDemoGraph();

Node\* startNode = testGraph.getNode(10.0, 20.0);

Node\* finishNode = testGraph.getNode(25.0, 35.0);

vector<Node\*> dfsPath = dfs(startNode, finishNode);

assert(dfsPath.size() >= 2 && "Путь DFS содержит менее двух узлов!");

assert(dfsPath.front() == startNode && "Первый узел пути DFS неправильный!");

assert(dfsPath.back() == finishNode && "Последний узел пути DFS неправильный!");

cout << "testDFSAlgorithm passed!\n";

}

void testDijkstraAlgorithm() {

Graph testGraph = createDemoGraph();

Node\* startNode = testGraph.getNode(10.0, 20.0);

Node\* finishNode = testGraph.getNode(25.0, 35.0);

vector<Node\*> dijkstraPath = dijkstra(testGraph, startNode, finishNode);

assert(dijkstraPath.size() >= 2 && "Путь Дейкстры содержит менее двух узлов!");

assert(dijkstraPath.front() == startNode && "Первый узел пути Дейкстры неправильный!");

assert(dijkstraPath.back() == finishNode && "Последний узел пути Дейкстры неправильный!");

cout << "testDijkstraAlgorithm passed!\n";

}

void testAStarAlgorithm() {

Graph testGraph = createDemoGraph();

Node\* startNode = testGraph.getNode(10.0, 20.0);

Node\* finishNode = testGraph.getNode(25.0, 35.0);

vector<Node\*> aStarPath = aStar(testGraph, startNode, finishNode);

assert(aStarPath.size() >= 2 && "Путь A\* содержит менее двух узлов!");

assert(aStarPath.front() == startNode && "Первый узел пути A\* неправильный!");

assert(aStarPath.back() == finishNode && "Последний узел пути A\* неправильный!");

cout << "testAStarAlgorithm passed!\n";

}

int main() {

// Создаём пустой граф и читаем из файла

Graph bigGraph;

bigGraph.readFromFile("C:/Users/Colbert/CLionProjects/untitled/spb\_graph.txt");

// Ищем ближайшие узлы

Node\* startNode = bigGraph.findClosestNode(60.017737, 30.385398);

Node\* endNode = bigGraph.findClosestNode(59.957238, 30.308108);

if (!startNode || !endNode) {

cerr << "Ошибка: не удалось найти стартовый или конечный узел.\n";

return 1;

}

cout << "Ближайший узел к старту: ("

<< startNode->lat << ", " << startNode->lon << ")\n";

cout << "Ближайший узел к финишу: ("

<< endNode->lat << ", " << endNode->lon << ")\n\n";

// Переменные для хранения времени и длины пути

long long BFS\_time = 0, DFS\_time = 0, Dijkstra\_time = 0, AStar\_time = 0;

size\_t BFS\_pathLen = 0, DFS\_pathLen = 0, Dijkstra\_pathLen = 0, AStar\_pathLen = 0;

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

// BFS

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

{

auto BFS\_timeBefore = high\_resolution\_clock::*now*();

vector<Node\*> pathBFS = bfs(startNode, endNode);

auto BFS\_timeAfter = high\_resolution\_clock::*now*();

bigGraph.printPath(pathBFS);

// Сохраняем метрику

BFS\_time = duration\_cast<milliseconds>(BFS\_timeAfter - BFS\_timeBefore).count();

BFS\_pathLen = pathBFS.size();

}

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

// DFS

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

{

auto DFS\_timeBefore = high\_resolution\_clock::*now*();

vector<Node\*> pathDFS = dfs(startNode, endNode);

auto DFS\_timeAfter = high\_resolution\_clock::*now*();

bigGraph.printPath(pathDFS);

DFS\_time = duration\_cast<milliseconds>(DFS\_timeAfter - DFS\_timeBefore).count();

DFS\_pathLen = pathDFS.size();

}

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

// Dijkstra

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

{

auto Dijkstra\_timeBefore = high\_resolution\_clock::*now*();

vector<Node\*> pathDijkstra = dijkstra(bigGraph, startNode, endNode);

auto Dijkstra\_timeAfter = high\_resolution\_clock::*now*();

bigGraph.printPath(pathDijkstra);

Dijkstra\_time = duration\_cast<milliseconds>(Dijkstra\_timeAfter - Dijkstra\_timeBefore).count();

Dijkstra\_pathLen = pathDijkstra.size();

}

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

// A\*

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

{

auto AStar\_timeBefore = high\_resolution\_clock::*now*();

vector<Node\*> pathAStar = aStar(bigGraph, startNode, endNode);

auto AStar\_timeAfter = high\_resolution\_clock::*now*();

bigGraph.printPath(pathAStar);

AStar\_time = duration\_cast<milliseconds>(AStar\_timeAfter - AStar\_timeBefore).count();

AStar\_pathLen = pathAStar.size();

}

cout << "\n==================== TESTS ====================\n";

// Запускаем тесты

testAddNodeFunction();

testGetNodeFunction();

testBFSAlgorithm();

testDFSAlgorithm();

testDijkstraAlgorithm();

testAStarAlgorithm();

cout << "\n";

cout << "All tests passed!\n";

// После тестов выводим суммарную информацию

cout << "\n=================== RESULTS ===================\n";

// BFS

cout << "BFS\n";

cout << "BFS Path Len: " << BFS\_pathLen << "\n";

cout << "BFS Time: " << BFS\_time << " ms\n\n";

// DFS

cout << "DFS\n";

cout << "DFS Path Len: " << DFS\_pathLen << "\n";

cout << "DFS Time: " << DFS\_time << " ms\n\n";

// Dijkstra

cout << "Dijkstra\n";

cout << "Dijkstra Path Len: " << Dijkstra\_pathLen << "\n";

cout << "Dijkstra Time: " << Dijkstra\_time << " ms\n\n";

// A\*

cout << "A\*\n";

cout << "A\* Path Len: " << AStar\_pathLen << "\n";

cout << "A\* Time: " << AStar\_time << " ms\n\n";

return 0;

}