

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8
«Поиск кратчайшего пути в графе»

Выполнил работу
Ширкунова Мария
Академическая группа №J3114
Принято
Дунаев Максим Владимирович

Санкт-Петербург

2024

Содержание отчета

1. Введение.....	3
2. Реализация	4
3. Анализ времени работы алгоритмов.....	13
4. Подсчет по времени и памяти.....	15
5. Заключение	20
6. Приложения	21

1. Введение

Цель работы: реализовать алгоритмы поиска кратчайшего пути между двумя вершинами графа.

Задачи:

- Реализовать алгоритмы BFS, DFS, A*, Дейкстры.
- Протестировать алгоритмы на некоторых точках карты спб.
- Посчитать время работы каждого из алгоритмов.
- Сравнить алгоритмы, проанализировать полученные результаты.

2. Реализация

Для начала импортируем необходимые для работы библиотеки:

- **vector**: Используется для хранения динамических массивов.
- **cmath**: Предоставляет функции для математических операций.
- **iostream**: Для ввода и вывода данных.
- **fstream**: Для работы с файлами.
- **sstream**: Для обработки строк как потоков.
- **chrono**: Для измерения времени выполнения алгоритмов.
- **queue**: Для реализации очередей, необходимых для BFS.
- **unordered_map**: Для хранения ассоциативных массивов (словарей).

```
#include <vector>
#include <cmath>
#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
#include <queue>
#include <unordered_map>
```

```
using namespace std;
```

Определим структуру вершины графа. Храним ширину и долготу.

```
struct Node {
    double lon, lat;
    vector<pair<Node*, double>> edges;
};
```

Определим структуру графа, храним вершины и ребра с весами. Используем данную в условии ЛР функцию нахождения ближайшей по расстоянию (весу) вершины графа. Парсим граф из txt файла. Строка "lon1,lat1:lon2,lat2,weight2;lon3,lat3,weight3" обозначает, что есть узел (lon1,lat1); и у него идут 2 ребра: ребро с весом weight2 в (lon2,lat2) и ребро с весом weight3 в (lon3,lat3).

```
struct Graph {
    vector<unique_ptr<Node>> nodes;
    unordered_map<string, Node*> node_map;

    Node* findClosestNode(double lat, double lon) {
        double minDistance = numeric_limits<double>::max();
        Node* closestNode = nullptr;
        for (const auto& node : nodes) {
            double distance = sqrt(pow(node->lat - lat, 2) + pow(node->lon - lon, 2));
            if (distance < minDistance) {
                minDistance = distance;
                closestNode = node.get();
            }
        }
    }
};
```

```

    }
    return closestNode;
}

Node* addNode(double lon, double lat) {
    auto vertex = make_unique<Node>(Node{ lon, lat });
    Node* vertex_ptr = vertex.get();
    nodes.push_back(move(vertex));
    ostringstream key_stream;
    key_stream << fixed << setprecision(10) << lon << "," << lat;
    string identifier = key_stream.str();
    node_map[identifier] = vertex_ptr;
    return vertex_ptr;
}

Node* getNode(double lon, double lat) {
    ostringstream key_stream;
    key_stream << fixed << setprecision(10) << lon << "," << lat;
    string identifier = key_stream.str();
    if (node_map.find(identifier) != node_map.end()) {
        return node_map[identifier];
    }
    return nullptr;
}

void parseFile(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        return;
    }
    string line;
    while (getline(file, line)) {
        processLine(line);
    }
}

void processLine(const string& line) {
    istringstream lineStream(line);
    string parentData;
    if (!getline(lineStream, parentData, ':')) {
        return;
    }
    auto [lon1, lat1] = extractCoordinates(parentData);
    Node* parentNode = getNode(lon1, lat1);
    if (!parentNode) {
        parentNode = addNode(lon1, lat1);
    }
    string edgesData;
    while (getline(lineStream, edgesData, ';')) {
        processEdgeData(edgesData, parentNode);
    }
}

pair<double, double> extractCoordinates(const string& data) {
    string cleanedData = data;
    replace(cleanedData.begin(), cleanedData.end(), ',', ' ');
    double lon = 0.0, lat = 0.0;
    istringstream coordStream(cleanedData);
    if (!(coordStream >> lon >> lat)) {
        return { 0.0, 0.0 };
    }
    return { lon, lat };
}

```

```

void processEdgeData(const string& edgeData, Node* parentNode) {
    double lon2, lat2, weight;
    string cleanedEdgeData = edgeData;
    replace(cleanedEdgeData.begin(), cleanedEdgeData.end(), ',', ' ');
    istringstream edgeStream(cleanedEdgeData);
    if (!(edgeStream >> lon2 >> lat2 >> weight)) {
        return;
    }
    Node* childNode = getNode(lon2, lat2);
    if (!childNode) {
        childNode = addNode(lon2, lat2);
    }
    parentNode->edges.emplace_back(childNode, weight);
    childNode->edges.emplace_back(parentNode, weight);
}
};

```

Реализуем алгоритм Дейкстры для поиска минимального по весу (кратчайшего пути) между двумя вершинами графа. Заводим карту расстояний от стартовой вершины до всех других и карту приоритетной очереди. Всем вершинам присваивается метка бесконечность. Стартовой вершине метка 0 – расстояние до нее равно нулю. Среди нерассмотренных вершин в приоритетной очереди, пополняемой соседями и сортируемой по весу ребер, находим вершину с наименьшей меткой. Для каждой необработанной вершины i , если путь к вершине j меньше существующей метки, заменить ее метку на новое расстояние. Продолжаем пока остаются необработанные вершины. Метка между искомыми вершинами окажется минимальным расстоянием. Восстанавливаем путь и возвращаем его.

```

vector<Node*> dijkstra(Graph& graph, Node* source, Node* destination) {
    unordered_map<Node*, Node*> predecessors;
    if (!source || !destination) return {};
    unordered_map<Node*, double> distances;
    for (const auto& vertex : graph.nodes)
        distances[vertex.get()] = numeric_limits<double>::infinity();
    distances[source] = 0.0;
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>>
priority_q;
    priority_q.emplace(0.0, source);
    while (!priority_q.empty()) {
        auto [current_distance, current_vertex] = priority_q.top();
        priority_q.pop();
        if (current_vertex == destination) {
            vector<Node*> route;
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first;
            double edge_weight = connection.second;
            double new_distance = current_distance + edge_weight;
            if (new_distance < distances[adjacent]) {
                distances[adjacent] = new_distance;
                predecessors[adjacent] = current_vertex;
                priority_q.emplace(new_distance, adjacent);
            }
        }
    }
}

```

```

    }
    return {};
}

```

Реализуем алгоритм поиска в ширину. В очередь помещаем стартовую вершину, заводим массив расстояний и храним путь. Пока в очереди есть необработанные вершины и текущая вершина не является целевой, помещаем соседей текущей вершины в очередь, обновляем расстояние и массив пути. Возвращаем путь.

```

vector<Node*> bfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    queue<Node*> q;
    unordered_map<Node*, Node*> predecessors;
    unordered_map<Node*, bool> visited;
    q.push(source);
    visited[source] = true;
    while (!q.empty()) {
        Node* current_vertex = q.front();
        q.pop();
        if (current_vertex == destination) {
            vector<Node*> route;
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first;
            if (!visited[adjacent]) {
                q.push(adjacent);
                visited[adjacent] = true;
                predecessors[adjacent] = current_vertex;
            }
        }
    }
    return {};
}

```

Реализуем алгоритм поиска в глубину. Аналогично BFS, заменяя структуру очереди на стек. Обновляем вектор пути, карту посещенных вершин. Если текущая вершина не целевая, то проходимся по еще не посещенным соседям, обновляя вес и пытаюсь найти один из всех существующих путей. Если же вершина целевая, то обновляем расстояние и возвращаем путь.

```

vector<Node*> dfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    unordered_map<Node*, bool> visited;
    unordered_map<Node*, Node*> predecessors;
    stack<Node*> s;
    s.push(source);
    visited[source] = true;
    while (!s.empty()) {
        Node* current_vertex = s.top();
        s.pop();
        if (current_vertex == destination) {
            vector<Node*> route;
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }

```

```

    }
    reverse(route.begin(), route.end());
    return route;
}
for (const auto& connection : current_vertex->edges) {
    Node* adjacent = connection.first;
    if (!visited[adjacent]) {
        s.push(adjacent);
        visited[adjacent] = true;
        predecessors[adjacent] = current_vertex;
    }
}
}
return {};
}

```

Реализуем алгоритм A*. Для начала заведем эвристику - минимальное расстояние между двумя вершинами. Заведем массивы f_score и g_score . В процессе работы алгоритма для вершин рассчитывается функция $f(v)=g(v)+h(v)$, где

- $g(v)$ — наименьшая стоимость пути в v из стартовой вершины,
- $h(v)$ — эвристическое приближение стоимости пути от v до конечной цели.

Заводим приоритетную очередь, карту, хранящую путь. Пока есть вершины, которые нужно обработать, и текущая вершина не является целевой, проходимся по соседним, рассчитываем эвристическое приближение, обновляем наименьшую стоимость пути при необходимости. Возвращаем найденный путь.

```

static double heuristic(Node* a, Node* b) {
    return sqrt(pow(a->lat - b->lat, 2) + pow(a->lon - b->lon, 2));
}

vector<Node*> aStar(Graph& graph, Node* closest_start, Node* closest_end) {
    if (!closest_start || !closest_end) {
        return {};
    }
    map<Node*, double> g_score;
    map<Node*, double> f_score;
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>> open_set;
    g_score[closest_start] = 0;
    f_score[closest_start] = heuristic(closest_start, closest_end);
    open_set.push({ f_score[closest_start], closest_start });
    map<Node*, Node*> came_from;
    while (!open_set.empty()) {
        auto current = open_set.top().second;
        open_set.pop();
        if (current == closest_end) {
            vector<Node*> path;
            for (Node* at = closest_end; at != nullptr; at = came_from[at]) {
                path.emplace_back(at);
            }
            reverse(path.begin(), path.end());
            return path;
        }
        for (const auto& neighbor : current->edges) {
            Node* neighbor_node = neighbor.first;
            double tentative_g_score = g_score[current] + neighbor.second;

```



```

        if (g_score.find(neighbor_node) == g_score.end() || tentative_g_score <
g_score[neighbor_node]) {
            came_from[neighbor_node] = current;
            g_score[neighbor_node] = tentative_g_score;
            f_score[neighbor_node] = g_score[neighbor_node] +
heuristic(neighbor_node, closest_end);
            open_set.push({ f_score[neighbor_node], neighbor_node });
        }
    }
}
return {};
}

```

Напишем тесты для функций.

```

void runTests() {
    // Test for BFS
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(1, 0);
        Node* c = graph.addNode(1, 1);
        Node* d = graph.addNode(0, 1);
        a->edges.emplace_back(b, 1);
        a->edges.emplace_back(d, 1);
        b->edges.emplace_back(c, 1);
        d->edges.emplace_back(c, 1);
        vector<Node*> path = bfs(a, c);
        assert(path.size() == 3 && path[0] == a && path[1] == b && path[2] == c);
    }

    // Test for DFS
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(1, 0);
        Node* c = graph.addNode(1, 1);
        Node* d = graph.addNode(0, 1);
        a->edges.emplace_back(b, 1);
        a->edges.emplace_back(d, 1);
        b->edges.emplace_back(c, 1);
        vector<Node*> path = dfs(a, c);
        assert(path.size() == 3 && path[0] == a && path[1] == b && path[2] == c);
    }

    // Test for Dijkstra's algorithm
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(2, 0);
        Node* c = graph.addNode(2, 2);
        a->edges.emplace_back(b, 4); // a - b (weight 4)
        a->edges.emplace_back(c, 2); // a - c (weight 2)
        b->edges.emplace_back(c, 1); // b - c (weight 1)
        vector<Node*> path = dijkstra(graph, a, c);
        assert(path.size() == 2 && path[0] == a && path[1] == c);
    }

    // Test for A*
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(2, 0);
        Node* c = graph.addNode(2, 2);
    }
}

```

```

    a->edges.emplace_back(b, 4); // a - b (weight 4)
    a->edges.emplace_back(c, 2); // a - c (weight 2)
    vector<Node*> path = aStar(graph, a, c);
    assert(path.size() == 2 && path[0] == a && path[1] == c);
}

cout << "All tests passed!" << endl;
}

```

Все тесты пройдены. Теперь рассчитаем время работы каждого из алгоритмов. Парсим граф из текстового файла, данного в условии ЛР. Заводим стартовую и целевую вершины – место жительства и университет ИТМО. Запускаем каждую из реализованных функций и подсчитываем время выполнения. Выводим на экран.

```

int main() {
    runTests();

    Graph graph;
    graph.parseFile("C:/Users/b0605/Downloads/spb_graph (2).txt");

    constexpr double start_lat = 59.848294; constexpr double start_lon = 30.329455;
    constexpr double goal_lat = 59.957238; constexpr double goal_lon = 30.308108;

    Node* start_node = graph.findClosestNode(start_lat, start_lon);
    Node* goal_node = graph.findClosestNode(goal_lat, goal_lon);

    // BFS
    cout << "BFS:" << endl;
    auto start_time = chrono::high_resolution_clock::now();
    const vector<Node*> bfs_path = bfs(start_node, goal_node);
    auto end_time = chrono::high_resolution_clock::now();
    double total_length = 0;
    for (size_t i = 0; i < bfs_path.size(); ++i) {
        //cout << "(" << bfs_path[i]->lat << ", " << bfs_path[i]->lon << ")";

        if (i < bfs_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : bfs_path[i]->edges) {
                if (edge.first == bfs_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << bfs_path.size() << endl;
    auto bfs_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of BFS: " << bfs_duration.count() << " ms" << endl;

    // Dijkstra
    cout << endl;
    cout << "Dijkstra:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> dijkstra_path = dijkstra(graph, start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < dijkstra_path.size(); ++i) {

```

```

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

        if (i < dijkstra_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : dijkstra_path[i]->edges) {
                if (edge.first == dijkstra_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << dijkstra_path.size() << endl;
    auto dijkstra_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of Dijkstra: " << dijkstra_duration.count() << " ms" << std::endl;

    // DFS
    cout << endl;
    cout << "DFS:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> dfs_path = dfs(start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < dfs_path.size(); ++i) {
        //cout << "(" << dfs_path[i]->lat << ", " << dfs_path[i]->lon << ")";

        if (i < dfs_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : dfs_path[i]->edges) {
                if (edge.first == dfs_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << dfs_path.size() << endl;
    auto dfs_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of DFS: " << dfs_duration.count() << " ms" << endl;

    // A*
    cout << endl;
    cout << "A*:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> a_star_path = aStar(graph, start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < a_star_path.size(); ++i) {
        //cout << "(" << a_star_path[i]->lat << ", " << a_star_path[i]->lon << ")";

        if (i < a_star_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : a_star_path[i]->edges) {
                if (edge.first == a_star_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
}

```

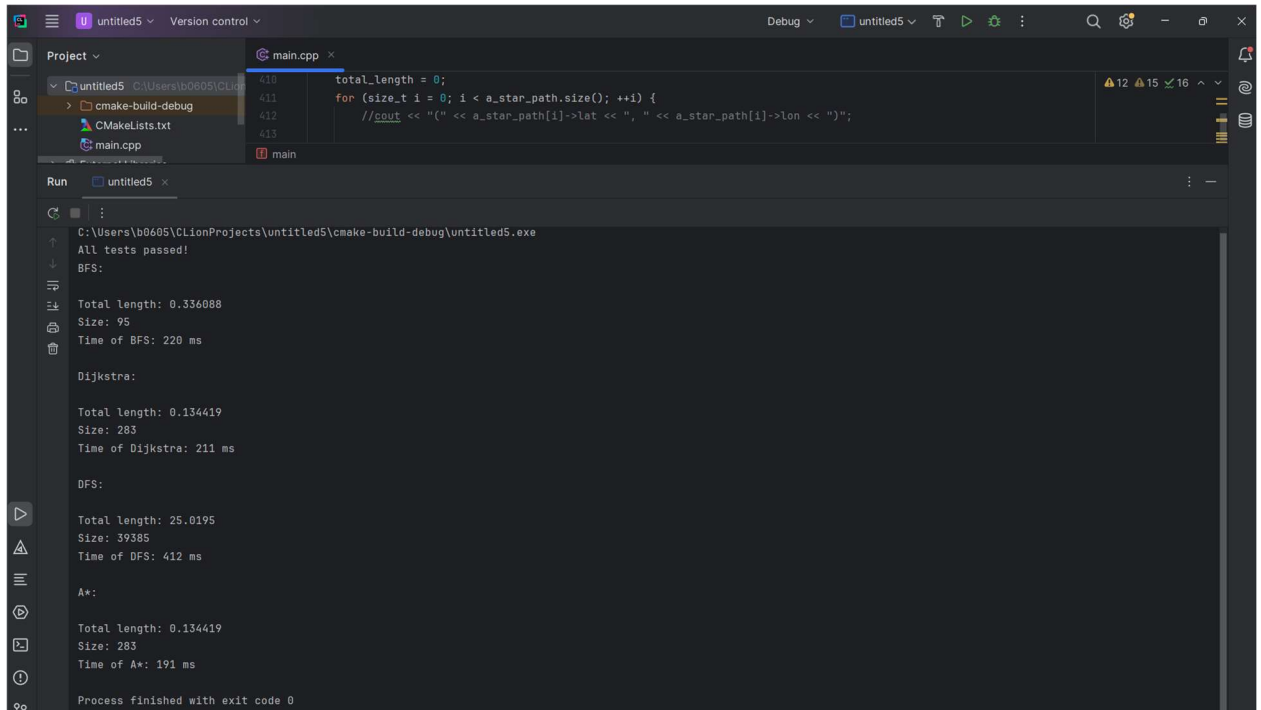
```

    }
}
cout << endl;
cout << "Total length: " << total_length << endl;
cout << "Size: " << a_star_path.size() << endl;
auto a_star_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
cout << "Time of A*: " << a_star_duration.count() << " ms" << endl;
}

```

3. Анализ времени работы алгоритмов.

После замера времени я получила следующие результаты:



```
untitled5 Version control
Project
  untitled5 C:\Users\b0605\CLionProjects\untitled5\cmake-build-debug
  CMakeLists.txt
  main.cpp
  main

main.cpp
410 total_length = 0;
411 for (size_t i = 0; i < a_star_path.size(); ++i) {
412     //cout << "(" << a_star_path[i]->lat << ", " << a_star_path[i]->lon << ")";
413 }

Run untitled5
C:\Users\b0605\CLionProjects\untitled5\cmake-build-debug\untitled5.exe
All tests passed!
BFS:
Total length: 0.336088
Size: 95
Time of BFS: 220 ms

Dijkstra:
Total length: 0.134419
Size: 283
Time of Dijkstra: 211 ms

DFS:
Total length: 25.0195
Size: 39385
Time of DFS: 412 ms

A*:
Total length: 0.134419
Size: 283
Time of A*: 191 ms

Process finished with exit code 0
```

```
C:\Users\b0605\CLionProjects\untitled5\cmake-build-debug\untitled5.exe
All tests passed!
BFS:

Total length: 0.336088
Size: 95
Time of BFS: 220 ms

Dijkstra:

Total length: 0.134419
Size: 283
Time of Dijkstra: 211 ms

DFS:

Total length: 25.0195
Size: 39385
Time of DFS: 412 ms

A*:

Total length: 0.134419
Size: 283
Time of A*: 191 ms

Process finished with exit code 0
```

Заметим, что быстрее всех был выполнен алгоритм A^* , затем Дейкстры, с небольшим отрывом BFS и самым медленным оказался DFS.

Почему так? **Алгоритм A^*** оказался самым быстрым, что объясняется его использованием эвристики, которая позволяет ему более эффективно исследовать пространство поиска. A^* выбирает путь с наименьшей оценкой стоимости, что позволяет ему избегать ненужных обходов. **Алгоритм Дейкстры** показал время, близкое к A^* , но немного медленнее. Это связано с тем, что Дейкстра исследует все возможные пути до достижения цели, не используя эвристики, что может привести к большому количеству проверок. **BFS** имеет время выполнения чуть больше, чем у Дейкстры. Этот алгоритм подходит для нахождения кратчайшего пути в невзвешенных графах, но в случае взвешенных графов его эффективность снижается, так как он не учитывает вес ребер и путь оказывается не минимальным. **DFS** показал наихудшее время выполнения среди всех алгоритмов, ответ так же неверен как и с BFS. Это связано с тем, что DFS не предназначен для нахождения кратчайшего пути и может исследовать множество ненужных ветвей графа, что значительно увеличивает время выполнения.

4. Подсчет по времени и памяти.

```
vector<Node*> bfs(Node* source, Node* destination) {
    if (!source || !destination) return {}; // O(1)
    queue<Node*> q; // O(1)
    unordered_map<Node*, Node*> predecessors; // O(1)
    unordered_map<Node*, bool> visited; // O(1)
    q.push(source); // O(1)
    visited[source] = true; // O(1)
    while (!q.empty()) { // O(V + E) (V - vertices, E - edges)
        Node* current_vertex = q.front(); // O(1)
        q.pop(); // O(1)
        if (current_vertex == destination) { // O(1)
            vector<Node*> route; // O(1)
            for (Node* at = destination; at != nullptr; at = predecessors[at]) { // O(V)
                route.push_back(at); // O(1)
            }
            reverse(route.begin(), route.end()); // O(V)
            return route; // O(1)
        }
        for (const auto& connection : current_vertex->edges) { // O(E)
            Node* adjacent = connection.first; // O(1)
            if (!visited[adjacent]) { // O(1)
                q.push(adjacent); // O(1)
                visited[adjacent] = true; // O(1)
                predecessors[adjacent] = current_vertex; // O(1)
            }
        }
    }
    return {}; // O(1)
}
// Итого: O((V+E)^2)
```

```
vector<Node*> bfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    queue<Node*> q; // V*16 байт
    unordered_map<Node*, Node*> predecessors; // V*32 байт
    unordered_map<Node*, bool> visited; // V*17 байт
    q.push(source);
    visited[source] = true;
    while (!q.empty()) {
        Node* current_vertex = q.front(); // 16 байт
        q.pop();
        if (current_vertex == destination) {
            vector<Node*> route; // V*16 байт
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first; // 16 байт
            if (!visited[adjacent]) {
                q.push(adjacent);
                visited[adjacent] = true;
                predecessors[adjacent] = current_vertex;
            }
        }
    }
    return {};
}
// Итого: V*(16+32+17+16) + 32 байт
```

```

vector<Node*> dfs(Node* source, Node* destination) {
    if (!source || !destination) return {}; // O(1)
    unordered_map<Node*, bool> visited; // O(1)
    unordered_map<Node*, Node*> predecessors; // O(1)
    stack<Node*> s; // O(1)
    s.push(source); // O(1)
    visited[source] = true; // O(1)
    while (!s.empty()) { // O(V + E)
        Node* current_vertex = s.top(); // O(1)
        s.pop(); // O(1)
        if (current_vertex == destination) { // O(1)
            vector<Node*> route; // O(1)
            for (Node* at = destination; at != nullptr; at = predecessors[at]) { // O(V)
                route.push_back(at); // O(1)
            }
            reverse(route.begin(), route.end()); // O(V)
            return route; // O(1)
        }
        for (const auto& connection : current_vertex->edges) { // O(E)
            Node* adjacent = connection.first; // O(1)
            if (!visited[adjacent]) { // O(1)
                s.push(adjacent); // O(1)
                visited[adjacent] = true; // O(1)
                predecessors[adjacent] = current_vertex; // O(1)
            }
        }
    }
    return {}; // O(1)
}
// Итого: O((V + E)^2)

vector<Node*> dfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    unordered_map<Node*, bool> visited; // V*17
    unordered_map<Node*, Node*> predecessors; // V*32
    stack<Node*> s; // V*16
    s.push(source);
    visited[source] = true;
    while (!s.empty()) {
        Node* current_vertex = s.top(); // 16
        s.pop();
        if (current_vertex == destination) {
            vector<Node*> route; // V*16
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first; // 16
            if (!visited[adjacent]) {
                s.push(adjacent);
                visited[adjacent] = true;
                predecessors[adjacent] = current_vertex;
            }
        }
    }
    return {};
}
// Итого: V*(17+32+16) + 32 байт

vector<Node*> dijkstra(Graph& graph, Node* source, Node* destination) {
    unordered_map<Node*, Node*> predecessors; // V*32

```



```

    if (!source || !destination) return {};
    unordered_map<Node*, double> distances; //V*24
    for (const auto& vertex : graph.nodes)
        distances[vertex.get()] = numeric_limits<double>::infinity();
    distances[source] = 0.0;
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>>
priority_q; // V*48
    priority_q.emplace(0.0, source);
    while (!priority_q.empty()) {
        auto [current_distance, current_vertex] = priority_q.top(); // V*32
        priority_q.pop();
        if (current_vertex == destination) {
            vector<Node*> route; // V*16
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first; // 16
            double edge_weight = connection.second; // 8
            double new_distance = current_distance + edge_weight; // 8
            if (new_distance < distances[adjacent]) {
                distances[adjacent] = new_distance;
                predecessors[adjacent] = current_vertex;
                priority_q.emplace(new_distance, adjacent);
            }
        }
    }
    return {};
}
// Итого: V*(32+24+48+32+16) + 32 байт

vector<Node*> dijkstra(Graph& graph, Node* source, Node* destination) {
    unordered_map<Node*, Node*> predecessors; // 0(1)
    if (!source || !destination) return {}; // 0(1)

    unordered_map<Node*, double> distances; // 0(1)

    for (const auto& vertex : graph.nodes) { // 0(V)
        distances[vertex.get()] = numeric_limits<double>::infinity(); // 0(1)
    }

    distances[source] = 0.0; // 0(V)

    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>>
priority_q; // 0(V)

    priority_q.emplace(0.0, source); // 0(log V)

    while (!priority_q.empty()) { // 0(E log V), так как мы можем извлечь все рёбра
        auto [current_distance, current_vertex] = priority_q.top(); // 0(log V)
        priority_q.pop(); // 0(log V)

        if (current_vertex == destination) { // 0(1)
            vector<Node*> route; // 0(V)

            for (Node* at = destination; at != nullptr; at = predecessors[at]) { //
0(V)
                route.push_back(at); // 0(1), добавление в вектор
            }

            reverse(route.begin(), route.end()); // 0(V)

```

```

        return route;    // O(1)
    }

    for (const auto& connection : current_vertex->edges) {    // O(E), обход всех
соседей
        Node* adjacent = connection.first;    // O(1)

        double edge_weight = connection.second;    // O(1)

        double new_distance = current_distance + edge_weight;    // O(1)

        if (new_distance < distances[adjacent]) {    // O(V), проверка расстояния
            distances[adjacent] = new_distance;    // O(1)

            predecessors[adjacent] = current_vertex;    // O(V)

            priority_q.emplace(new_distance, adjacent);    // O(log V), добавление в
приоритетную очередь
        }
    }

    return {};    // O(1)
}
// Итого: O(E log V)

vector<Node*> aStar(Graph& graph, Node* closest_start, Node* closest_end) {
    if (!closest_start || !closest_end) {    // O(1)
        return {};    // O(1)
    }

    map<Node*, double> g_score;    // O(V)
    map<Node*, double> f_score;    // O(V)

    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>> open_set;
// O(V)

    g_score[closest_start] = 0;    // O(V)

    f_score[closest_start] = heuristic(closest_start, closest_end);    // O(V)

    open_set.push({ f_score[closest_start], closest_start });    // O(log V)

    map<Node*, Node*> came_from;    // O(V)

    while (!open_set.empty()) {    // ~O(E log V), так как мы можем извлечь все рёбра
        auto current = open_set.top().second;    // O(log V)

        open_set.pop();    // O(log V)

        if (current == closest_end) {    // O(1)
            vector<Node*> path;    // O(V)

            for (Node* at = closest_end; at != nullptr; at = came_from[at]) {    // ~O(V),
для построения пути
                path.emplace_back(at);    // ~O(V), добавление в вектор
            }

            reverse(path.begin(), path.end());    // ~O(V)

            return path;    // ~O(1)
        }
    }
}

```

```

    for (const auto& neighbor : current->edges) { // ~O(E), обход всех соседей
        Node* neighbor_node = neighbor.first; // ~O(1)

        double tentative_g_score = g_score[current] + neighbor.second; // ~O(V)

        if (g_score.find(neighbor_node) == g_score.end() || tentative_g_score <
            g_score[neighbor_node]) { // ~O(V), проверка и обновление оценок g и f

            came_from[neighbor_node] = current; // ~O(V)

            g_score[neighbor_node] = tentative_g_score; // ~O(V)

            f_score[neighbor_node] = g_score[neighbor_node] +
                heuristic(neighbor_node, closest_end); // ~O(V)

            open_set.push({ f_score[neighbor_node], neighbor_node }); // ~O(log V),
добавление в приоритетную очередь
        }
    }

    return {}; // ~O(1)
}
// Итого: ~O(E log V)

vector<Node*> aStar(Graph& graph, Node* closest_start, Node* closest_end) {
    if (!closest_start || !closest_end) {
        return {};
    }
    map<Node*, double> g_score; // V*24
    map<Node*, double> f_score; // V*24
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>> open_set;
// V*48
    g_score[closest_start] = 0;
    f_score[closest_start] = heuristic(closest_start, closest_end);
    open_set.push({ f_score[closest_start], closest_start });
    map<Node*, Node*> came_from; // V*32
    while (!open_set.empty()) {
        auto current = open_set.top().second; // 16
        open_set.pop();
        if (current == closest_end) {
            vector<Node*> path; // V*16
            for (Node* at = closest_end; at != nullptr; at = came_from[at]) {
                path.emplace_back(at);
            }
            reverse(path.begin(), path.end());
            return path;
        }
        for (const auto& neighbor : current->edges) {
            Node* neighbor_node = neighbor.first; // 16
            double tentative_g_score = g_score[current] + neighbor.second; // 8
            if (g_score.find(neighbor_node) == g_score.end() || tentative_g_score <
                g_score[neighbor_node]) {
                came_from[neighbor_node] = current;
                g_score[neighbor_node] = tentative_g_score;
                f_score[neighbor_node] = g_score[neighbor_node] +
                    heuristic(neighbor_node, closest_end);
                open_set.push({ f_score[neighbor_node], neighbor_node });
            }
        }
    }
    return {};
}
// Итого: V*(48+48+32+16) + 40 байт

```

5. Заключение

В ходе выполнения лабораторной работы были реализованы алгоритмы поиска кратчайшего пути (минимального по сумме весов ребер в пути) между двумя вершинами графа: DFS, BFS, Дейкстры, A*.

Было измерено время выполнения каждого из алгоритмов на одинаковом входном наборе. Вследствие этого был сделан вывод о том, что для нахождения кратчайшего пути в взвешенных графах наиболее эффективными являются алгоритмы A* и Дейкстры. A* имеет явное преимущество благодаря своей эвристической составляющей, что делает его более быстрым в большинстве случаев.

6. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab-8.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <stack>
#include <cmath>
#include <limits>
#include <memory>
#include <algorithm>
#include <cassert>
#include <chrono>
#include <map>
#include <unordered_set>

using namespace std;

struct Node {
    double lon, lat;
    vector<pair<Node*, double>> edges;
};

struct Graph {
    vector<unique_ptr<Node>> nodes;
    unordered_map<string, Node*> node_map;

    Node* findClosestNode(double lat, double lon) {
        double minDistance = numeric_limits<double>::max();
        Node* closestNode = nullptr;
        for (const auto& node : nodes) {
            double distance = sqrt(pow(node->lat - lat, 2) + pow(node->lon - lon, 2));
            if (distance < minDistance) {
                minDistance = distance;
                closestNode = node.get();
            }
        }
        return closestNode;
    }

    Node* addNode(double lon, double lat) {
        auto vertex = make_unique<Node>(Node{ lon, lat });
        Node* vertex_ptr = vertex.get();
        nodes.push_back(move(vertex));
        ostringstream key_stream;
        key_stream << fixed << setprecision(10) << lon << "," << lat;
        string identifier = key_stream.str();
        node_map[identifier] = vertex_ptr;
        return vertex_ptr;
    }

    Node* getNode(double lon, double lat) {
        ostringstream key_stream;
        key_stream << fixed << setprecision(10) << lon << "," << lat;
        string identifier = key_stream.str();
        if (node_map.find(identifier) != node_map.end()) {
            return node_map[identifier];
        }
    }
};
```

```

    }
    return nullptr;
}

void parseFile(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        return;
    }
    string line;
    while (getline(file, line)) {
        processLine(line);
    }
}

void processLine(const string& line) {
    istringstream lineStream(line);
    string parentData;
    if (!getline(lineStream, parentData, ':')) {
        return;
    }
    auto [lon1, lat1] = extractCoordinates(parentData);
    Node* parentNode = getNode(lon1, lat1);
    if (!parentNode) {
        parentNode = addNode(lon1, lat1);
    }
    string edgesData;
    while (getline(lineStream, edgesData, ';')) {
        processEdgeData(edgesData, parentNode);
    }
}

pair<double, double> extractCoordinates(const string& data) {
    string cleanedData = data;
    replace(cleanedData.begin(), cleanedData.end(), ',', ' ');
    double lon = 0.0, lat = 0.0;
    istringstream coordStream(cleanedData);
    if (!(coordStream >> lon >> lat)) {
        return { 0.0, 0.0 };
    }
    return { lon, lat };
}

void processEdgeData(const string& edgeData, Node* parentNode) {
    double lon2, lat2, weight;
    string cleanedEdgeData = edgeData;
    replace(cleanedEdgeData.begin(), cleanedEdgeData.end(), ',', ' ');
    istringstream edgeStream(cleanedEdgeData);
    if (!(edgeStream >> lon2 >> lat2 >> weight)) {
        return;
    }
    Node* childNode = getNode(lon2, lat2);
    if (!childNode) {
        childNode = addNode(lon2, lat2);
    }
    parentNode->edges.emplace_back(childNode, weight);
    childNode->edges.emplace_back(parentNode, weight);
}

};

vector<Node*> bfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    queue<Node*> q;
    unordered_map<Node*, Node*> predecessors;

```

```

unordered_map<Node*, bool> visited;
q.push(source);
visited[source] = true;
while (!q.empty()) {
    Node* current_vertex = q.front();
    q.pop();
    if (current_vertex == destination) {
        vector<Node*> route;
        for (Node* at = destination; at != nullptr; at = predecessors[at]) {
            route.push_back(at);
        }
        reverse(route.begin(), route.end());
        return route;
    }
    for (const auto& connection : current_vertex->edges) {
        Node* adjacent = connection.first;
        if (!visited[adjacent]) {
            q.push(adjacent);
            visited[adjacent] = true;
            predecessors[adjacent] = current_vertex;
        }
    }
}
return {};
}

vector<Node*> dfs(Node* source, Node* destination) {
    if (!source || !destination) return {};
    unordered_map<Node*, bool> visited;
    unordered_map<Node*, Node*> predecessors;
    stack<Node*> s;
    s.push(source);
    visited[source] = true;
    while (!s.empty()) {
        Node* current_vertex = s.top();
        s.pop();
        if (current_vertex == destination) {
            vector<Node*> route;
            for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                route.push_back(at);
            }
            reverse(route.begin(), route.end());
            return route;
        }
        for (const auto& connection : current_vertex->edges) {
            Node* adjacent = connection.first;
            if (!visited[adjacent]) {
                s.push(adjacent);
                visited[adjacent] = true;
                predecessors[adjacent] = current_vertex;
            }
        }
    }
    return {};
}

vector<Node*> dijkstra(Graph& graph, Node* source, Node* destination) {
    unordered_map<Node*, Node*> predecessors;
    if (!source || !destination) return {};
    unordered_map<Node*, double> distances;
    for (const auto& vertex : graph.nodes)
        distances[vertex.get()] = numeric_limits<double>::infinity();
    distances[source] = 0.0;
}

```

```

        priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>>
priority_q;
        priority_q.emplace(0.0, source);
        while (!priority_q.empty()) {
            auto [current_distance, current_vertex] = priority_q.top();
            priority_q.pop();
            if (current_vertex == destination) {
                vector<Node*> route;
                for (Node* at = destination; at != nullptr; at = predecessors[at]) {
                    route.push_back(at);
                }
                reverse(route.begin(), route.end());
                return route;
            }
            for (const auto& connection : current_vertex->edges) {
                Node* adjacent = connection.first;
                double edge_weight = connection.second;
                double new_distance = current_distance + edge_weight;
                if (new_distance < distances[adjacent]) {
                    distances[adjacent] = new_distance;
                    predecessors[adjacent] = current_vertex;
                    priority_q.emplace(new_distance, adjacent);
                }
            }
        }
        return {};
    }

static double heuristic(Node* a, Node* b) {
    return sqrt(pow(a->lat - b->lat, 2) + pow(a->lon - b->lon, 2));
}

vector<Node*> aStar(Graph& graph, Node* closest_start, Node* closest_end) {
    if (!closest_start || !closest_end) {
        return {};
    }
    map<Node*, double> g_score;
    map<Node*, double> f_score;
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, greater<>> open_set;
    g_score[closest_start] = 0;
    f_score[closest_start] = heuristic(closest_start, closest_end);
    open_set.push({ f_score[closest_start], closest_start });
    map<Node*, Node*> came_from;
    while (!open_set.empty()) {
        auto current = open_set.top().second;
        open_set.pop();
        if (current == closest_end) {
            vector<Node*> path;
            for (Node* at = closest_end; at != nullptr; at = came_from[at]) {
                path.emplace_back(at);
            }
            reverse(path.begin(), path.end());
            return path;
        }
        for (const auto& neighbor : current->edges) {
            Node* neighbor_node = neighbor.first;
            double tentative_g_score = g_score[current] + neighbor.second;
            if (g_score.find(neighbor_node) == g_score.end() || tentative_g_score <
g_score[neighbor_node]) {
                came_from[neighbor_node] = current;
                g_score[neighbor_node] = tentative_g_score;
                f_score[neighbor_node] = g_score[neighbor_node] +
heuristic(neighbor_node, closest_end);
                open_set.push({ f_score[neighbor_node], neighbor_node });
            }
        }
    }
}

```



```

    }
    }
}
return {};
}

void runTests() {
    // Test for BFS
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(1, 0);
        Node* c = graph.addNode(1, 1);
        Node* d = graph.addNode(0, 1);
        a->edges.emplace_back(b, 1);
        a->edges.emplace_back(d, 1);
        b->edges.emplace_back(c, 1);
        d->edges.emplace_back(c, 1);
        vector<Node*> path = bfs(a, c);
        assert(path.size() == 3 && path[0] == a && path[1] == b && path[2] == c);
    }

    // Test for DFS
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(1, 0);
        Node* c = graph.addNode(1, 1);
        Node* d = graph.addNode(0, 1);
        a->edges.emplace_back(b, 1);
        a->edges.emplace_back(d, 1);
        b->edges.emplace_back(c, 1);
        vector<Node*> path = dfs(a, c);
        assert(path.size() == 3 && path[0] == a && path[1] == b && path[2] == c);
    }

    // Test for Dijkstra's algorithm
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(2, 0);
        Node* c = graph.addNode(2, 2);
        a->edges.emplace_back(b, 4); // a - b (weight 4)
        a->edges.emplace_back(c, 2); // a - c (weight 2)
        b->edges.emplace_back(c, 1); // b - c (weight 1)
        vector<Node*> path = dijkstra(graph, a, c);
        assert(path.size() == 2 && path[0] == a && path[1] == c);
    }

    // Test for A*
    {
        Graph graph;
        Node* a = graph.addNode(0, 0);
        Node* b = graph.addNode(2, 0);
        Node* c = graph.addNode(2, 2);
        a->edges.emplace_back(b, 4); // a - b (weight 4)
        a->edges.emplace_back(c, 2); // a - c (weight 2)
        vector<Node*> path = aStar(graph, a, c);
        assert(path.size() == 2 && path[0] == a && path[1] == c);
    }

    cout << "All tests passed!" << endl;
}

```

```

int main() {
    runTests();

    Graph graph;
    graph.parseFile("spb_graph.txt");

    constexpr double start_lat = 59.848294; constexpr double start_lon = 30.329455;
    constexpr double goal_lat = 59.957238; constexpr double goal_lon = 30.308108;

    Node* start_node = graph.findClosestNode(start_lat, start_lon);
    Node* goal_node = graph.findClosestNode(goal_lat, goal_lon);

    // BFS
    cout << "BFS:" << endl;
    auto start_time = chrono::high_resolution_clock::now();
    const vector<Node*> bfs_path = bfs(start_node, goal_node);
    auto end_time = chrono::high_resolution_clock::now();
    double total_length = 0;
    for (size_t i = 0; i < bfs_path.size(); ++i) {
        //cout << "(" << bfs_path[i]->lat << ", " << bfs_path[i]->lon << ")";

        if (i < bfs_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : bfs_path[i]->edges) {
                if (edge.first == bfs_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << bfs_path.size() << endl;
    auto bfs_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of BFS: " << bfs_duration.count() << " ms" << endl;

    // Dijkstra
    cout << endl;
    cout << "Dijkstra:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> dijkstra_path = dijkstra(graph, start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < dijkstra_path.size(); ++i) {
        //cout << "(" << dijkstra_path[i]->lat << ", " << dijkstra_path[i]->lon << ")";

        if (i < dijkstra_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : dijkstra_path[i]->edges) {
                if (edge.first == dijkstra_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << dijkstra_path.size() << endl;

```

```

    auto dijkstra_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of Dijkstra: " << dijkstra_duration.count() << " ms" << std::endl;

    // DFS
    cout << endl;
    cout << "DFS:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> dfs_path = dfs(start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < dfs_path.size(); ++i) {
        //cout << "(" << dfs_path[i]->lat << ", " << dfs_path[i]->lon << ")";

        if (i < dfs_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : dfs_path[i]->edges) {
                if (edge.first == dfs_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << dfs_path.size() << endl;
    auto dfs_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of DFS: " << dfs_duration.count() << " ms" << endl;

    // A*
    cout << endl;
    cout << "A*:" << endl;
    start_time = chrono::high_resolution_clock::now();
    const vector<Node*> a_star_path = aStar(graph, start_node, goal_node);
    end_time = chrono::high_resolution_clock::now();
    total_length = 0;
    for (size_t i = 0; i < a_star_path.size(); ++i) {
        //cout << "(" << a_star_path[i]->lat << ", " << a_star_path[i]->lon << ")";

        if (i < a_star_path.size() - 1) {
            //cout << " - ";
            for (const auto& edge : a_star_path[i]->edges) {
                if (edge.first == a_star_path[i + 1]) {
                    total_length += edge.second;
                    break;
                }
            }
        }
    }
    cout << endl;
    cout << "Total length: " << total_length << endl;
    cout << "Size: " << a_star_path.size() << endl;
    auto a_star_duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
    cout << "Time of A*: " << a_star_duration.count() << " ms" << endl;
}

```