

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.....	10
4. Заключение	12
5. Приложения	13

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<Node*> nodes;

    Node* find_closest_node(double lat, double lon) {
        Node* closest_node = nullptr;
        double min_weight = numeric_limits<double>::max();

        for (auto node : nodes) {
            for (auto& edge : node->edges) {
                auto [neighbor, weight] = edge;
                if (weight < min_weight) {
                    closest_node = neighbor;
                    min_weight = weight;
                }
            }
        }
    }
};
```

```

    }
    }
    }
    return closest_node;
}

void load_from_file(const string& filename) {
    ifstream file(filename);
    string line;
    while (getline(file, line)) {
        istringstream iss(line);
        string node_info;
        getline(iss, node_info, ':');
        double lon1, lat1;
        sscanf(node_info.c_str(), "%lf,%lf", &lon1, &lat1);
        Node* node = new Node{ lon1, lat1 };
        string edge_info;
        while (getline(iss, edge_info, ';')) {
            if (edge_info.empty()) continue;
            double lon2, lat2, weight;
            sscanf(edge_info.c_str(), "%lf,%lf,%lf", &lon2, &lat2, &weight);
            Node* neighbor = new Node{ lon2, lat2 };
            node->edges.emplace_back(neighbor, weight);
        }
        nodes.push_back(node);
    }
}
};

```

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* start, Node* goal) {
    unordered_map<Node*, double> distances;
    unordered_map<Node*, Node*> previous;

    auto cmp = [](pair<double, Node*> left, pair<double, Node*> right) { return
left.first > right.first; };
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, decltype(cmp)>
queue(cmp);

    for (auto node : graph.nodes) {
        distances[node] = numeric_limits<double>::max();
    }

    distances[start] = 0;
    queue.push({ 0, start });

    while (!queue.empty()) {
        auto [current_distance, current_node] = queue.top();

```

```

queue.pop();

if (current_node == goal) break;

for (auto& edge : current_node->edges) {
    auto [neighbor, weight] = edge;
    double new_distance = current_distance + weight;

    if (new_distance < distances[neighbor]) {
        distances[neighbor] = new_distance;
        previous[neighbor] = current_node;
        queue.push({ new_distance, neighbor });
    }
}

vector<Node*> path;
for (Node* at = goal; at != nullptr; at = previous[at]) {
    path.push_back(at);
}
reverse(path.begin(), path.end());

return path;
}

```

Реализуем алгоритм поиска в ширину. В очередь помещаем стартовую вершину, заводим массив расстояний и храним путь в `came_from`. 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(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, Node*> came_from;
    unordered_map<Node*, double> path_cost;
    queue<Node*> queue;

    queue.push(start);
    came_from[start] = nullptr;
    path_cost[start] = 0.0;

    while (!queue.empty()) {
        Node* current = queue.front();
        queue.pop();

        if (current == goal) break;

        for (auto& edge : current->edges) {
            Node* neighbor = edge.first;
            double weight = edge.second;

            if (came_from.find(neighbor) == came_from.end()) {
                queue.push(neighbor);
                came_from[neighbor] = current;
                path_cost[neighbor] = path_cost[current] + weight;
            }
        }
    }

    vector<Node*> path;
    for (Node* at = goal; at != nullptr; at = came_from[at]) {
        path.push_back(at);
    }
}

```

```

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

    return path;
}

```

Реализуем рекурсивный алгоритм поиска в глубину. Обновляем вектор пути, мапу посещенных вершин. Если текущая вершина не целевая, то проходимся по еще не посещенным соседям рекурсивно, обновляя вес и пытаясь найти минимальный из всех существующих путей. Если же вершина целевая, то обновляем расстояние и возвращаем путь.

```

void dfs_recursive(Node* current, Node* goal,
    unordered_map<Node*, bool>& visited,
    vector<Node*>& path,
    bool& found,
    double& total_weight,
    double current_weight) {

    if (found) return;

    visited[current] = true;
    path.push_back(current);

    if (current == goal) {
        found = true;
        total_weight += current_weight;
        return;
    }

    for (auto& edge : current->edges) {
        Node* neighbor = edge.first;
        double weight = edge.second;

        if (!visited[neighbor]) {
            dfs_recursive(neighbor, goal, visited, path, found,
                total_weight,
                current_weight + weight);
        }
    }

    if (!found) {
        path.pop_back();
    }
}

```

Реализуем вывод минимального пути поиска в глубину.

```

vector<Node*> dfs(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, bool> visited;
    vector<Node*> path;

    bool found = false;
    double total_weight = 0.0;

    dfs_recursive(start, goal, visited, path, found,
        total_weight,
        0.0);

    return path;
}

```

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.

```
double heuristic(Node* a, Node* b) {
    for (auto& edge : a->edges) {
        if (edge.first == b) {
            return edge.second;
        }
    }
    return numeric_limits<double>::max();
}

vector<Node*> a_star(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, double> g_score;
    unordered_map<Node*, double> f_score;

    auto cmp = [](pair<double, Node*> left,
                  pair<double, Node*> right) { return left.first > right.first; };

    priority_queue<pair<double, Node*>,
                  vector<pair<double, Node*>>,
                  decltype(cmp)> open_set(cmp);

    g_score[start] = 0;
    f_score[start] = heuristic(start, goal);

    open_set.push({ f_score[start], start });
    unordered_map<Node*, Node*> came_from;

    while (!open_set.empty()) {
        auto [_, current] = open_set.top();
        open_set.pop();

        if (current == goal) break;

        for (auto& edge : current->edges) {
            auto [neighbor, weight] = edge;

            double tentative_g_score = g_score[current] + weight;

            if (g_score.find(neighbor) == g_score.end()) {
                g_score[neighbor] = numeric_limits<double>::max();
            }

            if (tentative_g_score < g_score[neighbor]) {
                came_from[neighbor] = current;
                g_score[neighbor] = tentative_g_score;
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal);
            }
        }
    }
}
```



```

        open_set.push({ f_score[neighbor], neighbor });
    }
}

vector<Node*> path;

if (came_from.find(goal) != came_from.end()) {
    for (Node* at = goal; at != nullptr; at = came_from[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());
}

return path;
}

```

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() {
    Graph graph;
    graph.load_from_file("spb_graph.txt");

    Node* start_node = graph.find_closest_node(59.84829416, 30.329455);
    Node* goal_node = graph.find_closest_node(59.95672871, 30.309411);

    auto start_time_dijkstra = chrono::high_resolution_clock::now();
    auto dijkstra_path = dijkstra(graph, start_node, goal_node);
    auto end_time_dijkstra = chrono::high_resolution_clock::now();
    auto elapsed_time_dijkstra =
        chrono::duration_cast<chrono::duration<double>>(end_time_dijkstra - start_time_dijkstra);
    cout << "Time of Dijkstra: " << elapsed_time_dijkstra.count() << " seconds\n";

    auto start_time_bfs = chrono::high_resolution_clock::now();
    auto bfs_path = bfs(graph, start_node, goal_node);
    auto end_time_bfs = chrono::high_resolution_clock::now();
    auto elapsed_time_bfs = chrono::duration_cast<chrono::duration<double>>(end_time_bfs
- start_time_bfs);
    cout << "Time of BFS: " << elapsed_time_bfs.count() << " seconds\n";

    auto start_time_dfs = chrono::high_resolution_clock::now();
    auto dfs_path = dfs(graph, start_node, goal_node);
    auto end_time_dfs = chrono::high_resolution_clock::now();
    auto elapsed_time_dfs = chrono::duration_cast<chrono::duration<double>>(end_time_dfs
- start_time_dfs);
    cout << "Time of DFS: " << elapsed_time_dfs.count() << " seconds\n";

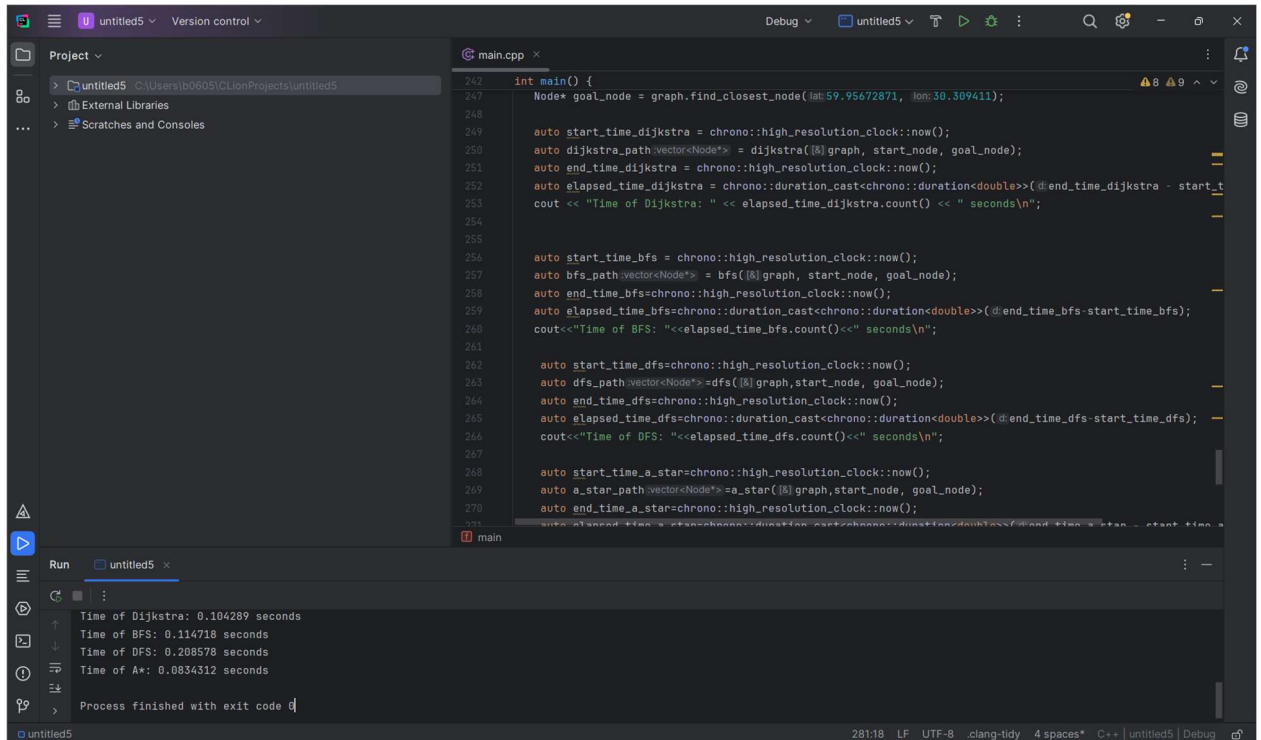
    auto start_time_a_star = chrono::high_resolution_clock::now();
    auto a_star_path = a_star(graph, start_node, goal_node);
    auto end_time_a_star = chrono::high_resolution_clock::now();
    auto elapsed_time_a_star =
        chrono::duration_cast<chrono::duration<double>>(end_time_a_star - start_time_a_star);
    cout << "Time of A*: " << elapsed_time_a_star.count() << " seconds\n";

    return 0;
}

```

3. Анализ времени работы алгоритмов.

После замера времени я получила следующие результаты:

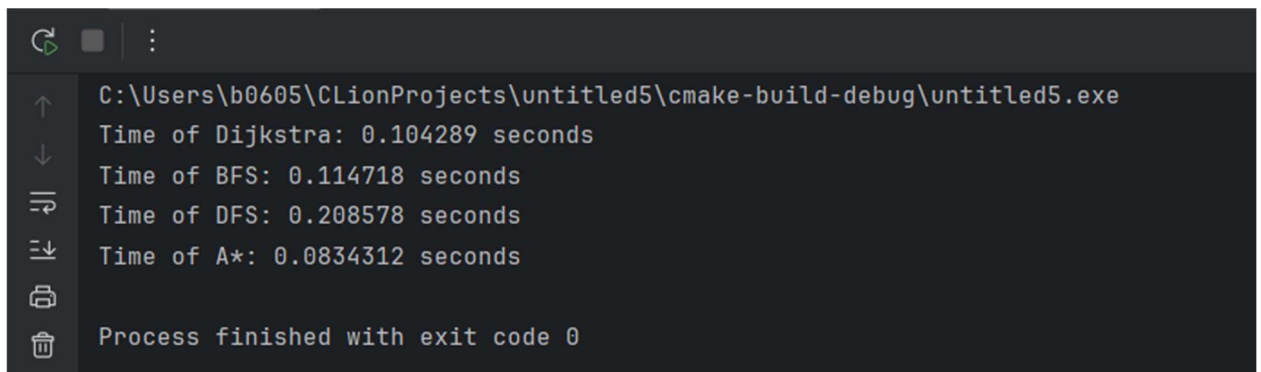


The screenshot shows the CLion IDE interface. The main editor displays a C++ file named `main.cpp` with the following code:

```
242 int main() {
243     Node* goal_node = graph.find_closest_node(lat:59.95672871, lon:30.309411);
244
245     auto start_time_dijkstra = chrono::high_resolution_clock::now();
246     auto dijkstra_path = dijkstra(&graph, start_node, goal_node);
247     auto end_time_dijkstra = chrono::high_resolution_clock::now();
248     auto elapsed_time_dijkstra = chrono::duration_cast<chrono::duration<double>>(&end_time_dijkstra - start_time_dijkstra);
249     cout << "Time of Dijkstra: " << elapsed_time_dijkstra.count() << " seconds\n";
250
251     auto start_time_bfs = chrono::high_resolution_clock::now();
252     auto bfs_path = bfs(&graph, start_node, goal_node);
253     auto end_time_bfs = chrono::high_resolution_clock::now();
254     auto elapsed_time_bfs = chrono::duration_cast<chrono::duration<double>>(&end_time_bfs - start_time_bfs);
255     cout << "Time of BFS: " << elapsed_time_bfs.count() << " seconds\n";
256
257     auto start_time_dfs = chrono::high_resolution_clock::now();
258     auto dfs_path = dfs(&graph, start_node, goal_node);
259     auto end_time_dfs = chrono::high_resolution_clock::now();
260     auto elapsed_time_dfs = chrono::duration_cast<chrono::duration<double>>(&end_time_dfs - start_time_dfs);
261     cout << "Time of DFS: " << elapsed_time_dfs.count() << " seconds\n";
262
263     auto start_time_a_star = chrono::high_resolution_clock::now();
264     auto a_star_path = a_star(&graph, start_node, goal_node);
265     auto end_time_a_star = chrono::high_resolution_clock::now();
266     auto elapsed_time_a_star = chrono::duration_cast<chrono::duration<double>>(&end_time_a_star - start_time_a_star);
267     cout << "Time of A*: " << elapsed_time_a_star.count() << " seconds\n";
268 }
```

The Run console at the bottom shows the following output:

```
Time of Dijkstra: 0.104289 seconds
Time of BFS: 0.114718 seconds
Time of DFS: 0.208578 seconds
Time of A*: 0.0834312 seconds
Process finished with exit code 0
```



The screenshot shows the Run console output, which is a copy of the output from the previous screenshot:

```
C:\Users\b0605\CLionProjects\untitled5\cmake-build-debug\untitled5.exe
Time of Dijkstra: 0.104289 seconds
Time of BFS: 0.114718 seconds
Time of DFS: 0.208578 seconds
Time of A*: 0.0834312 seconds
Process finished with exit code 0
```

Заметим, что быстрее всех был выполнен алгоритм A*, затем Дейкстры, с большим 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 большему 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 теперь учитывает вес ребер. **DFS** показал наихудшее время выполнения среди всех алгоритмов. 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. 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.

5. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab-8.cpp

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

using namespace std;

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

struct Graph {
    vector<Node*> nodes;

    Node* find_closest_node(double lat, double lon) {
        Node* closest_node = nullptr;
        double min_weight = numeric_limits<double>::max();

        for (auto node : nodes) {
            for (auto& edge : node->edges) {
                auto [neighbor, weight] = edge;
                if (weight < min_weight) {
                    closest_node = neighbor;
                    min_weight = weight;
                }
            }
        }
        return closest_node;
    }

    void load_from_file(const string& filename) {
        ifstream file(filename);
        string line;
        while (getline(file, line)) {
            istringstream iss(line);
            string node_info;
            getline(iss, node_info, ':');
            double lon1, lat1;
            sscanf(node_info.c_str(), "%lf,%lf", &lon1, &lat1);
            Node* node = new Node{ lon1, lat1 };
            string edge_info;
            while (getline(iss, edge_info, ';')) {
                if (edge_info.empty()) continue;
                double lon2, lat2, weight;
                sscanf(edge_info.c_str(), "%lf,%lf,%lf", &lon2, &lat2, &weight);
                Node* neighbor = new Node{ lon2, lat2 };
                node->edges.emplace_back(neighbor, weight);
            }
            nodes.push_back(node);
        }
    }
};
```

```

vector<Node*> dijkstra(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, double> distances;
    unordered_map<Node*, Node*> previous;

    auto cmp = [](pair<double, Node*> left, pair<double, Node*> right) { return
left.first > right.first; };
    priority_queue<pair<double, Node*>, vector<pair<double, Node*>>, decltype(cmp)>
queue(cmp);

    for (auto node : graph.nodes) {
        distances[node] = numeric_limits<double>::max();
    }

    distances[start] = 0;
    queue.push({ 0, start });

    while (!queue.empty()) {
        auto [current_distance, current_node] = queue.top();
        queue.pop();

        if (current_node == goal) break;

        for (auto& edge : current_node->edges) {
            auto [neighbor, weight] = edge;
            double new_distance = current_distance + weight;

            if (new_distance < distances[neighbor]) {
                distances[neighbor] = new_distance;
                previous[neighbor] = current_node;
                queue.push({ new_distance, neighbor });
            }
        }
    }

    vector<Node*> path;
    for (Node* at = goal; at != nullptr; at = previous[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());

    return path;
}

```

```

vector<Node*> bfs(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, Node*> came_from;
    unordered_map<Node*, double> path_cost;
    queue<Node*> queue;

    queue.push(start);
    came_from[start] = nullptr;
    path_cost[start] = 0.0;

    while (!queue.empty()) {
        Node* current = queue.front();
        queue.pop();

        if (current == goal) break;

        for (auto& edge : current->edges) {
            Node* neighbor = edge.first;
            double weight = edge.second;

```

```

        if (came_from.find(neighbor) == came_from.end()) {
            queue.push(neighbor);
            came_from[neighbor] = current;
            path_cost[neighbor] = path_cost[current] + weight;
        }
    }
}

vector<Node*> path;
for (Node* at = goal; at != nullptr; at = came_from[at]) {
    path.push_back(at);
}

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

return path;
}

void dfs_recursive(Node* current, Node* goal,
    unordered_map<Node*, bool>& visited,
    vector<Node*>& path,
    bool& found,
    double& total_weight,
    double current_weight) {

    if (found) return;

    visited[current] = true;
    path.push_back(current);

    if (current == goal) {
        found = true;
        total_weight += current_weight;
        return;
    }

    for (auto& edge : current->edges) {
        Node* neighbor = edge.first;
        double weight = edge.second;

        if (!visited[neighbor]) {
            dfs_recursive(neighbor, goal, visited, path, found,
                total_weight,
                current_weight + weight);
        }
    }

    if (!found) {
        path.pop_back();
    }
}

vector<Node*> dfs(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, bool> visited;
    vector<Node*> path;

    bool found = false;
    double total_weight = 0.0;

    dfs_recursive(start, goal, visited, path, found,
        total_weight,
        0.0);

    return path;
}

```

```

}

double heuristic(Node* a, Node* b) {
    for (auto& edge : a->edges) {
        if (edge.first == b) {
            return edge.second;
        }
    }
    return numeric_limits<double>::max();
}

vector<Node*> a_star(Graph& graph, Node* start, Node* goal) {
    unordered_map<Node*, double> g_score;
    unordered_map<Node*, double> f_score;

    auto cmp = [](pair<double, Node*> left,
                  pair<double, Node*> right) { return left.first > right.first; };

    priority_queue<pair<double, Node*>,
                  vector<pair<double, Node*>>,
                  decltype(cmp)> open_set(cmp);

    g_score[start] = 0;
    f_score[start] = heuristic(start, goal);

    open_set.push({ f_score[start], start });
    unordered_map<Node*, Node*> came_from;

    while (!open_set.empty()) {
        auto [_, current] = open_set.top();
        open_set.pop();

        if (current == goal) break;

        for (auto& edge : current->edges) {
            auto [neighbor, weight] = edge;

            double tentative_g_score = g_score[current] + weight;

            if (g_score.find(neighbor) == g_score.end()) {
                g_score[neighbor] = numeric_limits<double>::max();
            }

            if (tentative_g_score < g_score[neighbor]) {
                came_from[neighbor] = current;
                g_score[neighbor] = tentative_g_score;
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal);
                open_set.push({ f_score[neighbor], neighbor });
            }
        }
    }

    vector<Node*> path;

    if (came_from.find(goal) != came_from.end()) {
        for (Node* at = goal; at != nullptr; at = came_from[at]) {
            path.push_back(at);
        }
        reverse(path.begin(), path.end());
    }

    return path;
}

```



```

int main() {
    Graph graph;
    graph.load_from_file("spb_graph.txt");

    Node* start_node = graph.find_closest_node(59.84829416, 30.329455);
    Node* goal_node = graph.find_closest_node(59.95672871, 30.309411);

    auto start_time_dijkstra = chrono::high_resolution_clock::now();
    auto dijkstra_path = dijkstra(graph, start_node, goal_node);
    auto end_time_dijkstra = chrono::high_resolution_clock::now();
    auto elapsed_time_dijkstra =
chrono::duration_cast<chrono::duration<double>>(end_time_dijkstra - start_time_dijkstra);
    cout << "Time of Dijkstra: " << elapsed_time_dijkstra.count() << " seconds\n";

    auto start_time_bfs = chrono::high_resolution_clock::now();
    auto bfs_path = bfs(graph, start_node, goal_node);
    auto end_time_bfs = chrono::high_resolution_clock::now();
    auto elapsed_time_bfs = chrono::duration_cast<chrono::duration<double>>(end_time_bfs
- start_time_bfs);
    cout << "Time of BFS: " << elapsed_time_bfs.count() << " seconds\n";

    auto start_time_dfs = chrono::high_resolution_clock::now();
    auto dfs_path = dfs(graph, start_node, goal_node);
    auto end_time_dfs = chrono::high_resolution_clock::now();
    auto elapsed_time_dfs = chrono::duration_cast<chrono::duration<double>>(end_time_dfs
- start_time_dfs);
    cout << "Time of DFS: " << elapsed_time_dfs.count() << " seconds\n";

    auto start_time_a_star = chrono::high_resolution_clock::now();
    auto a_star_path = a_star(graph, start_node, goal_node);
    auto end_time_a_star = chrono::high_resolution_clock::now();
    auto elapsed_time_a_star =
chrono::duration_cast<chrono::duration<double>>(end_time_a_star - start_time_a_star);
    cout << "Time of A*: " << elapsed_time_a_star.count() << " seconds\n";

    return 0;
}

```