ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Поиск кратчайшего пути в графе»

Выполнила работу

Луценко Елена

Академическая группа №J3112

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

1. Введение

Цель работы: найти путь между двумя вершинами графа (от дома до места учёбы) с помощью различных алгоритмов и сравнить их результаты.

Задачи:

* Реализовать алгоритм BFS
* Реализовать алгоритм DFS
* Реализовать алгоритм Dijkstra
* Реализовать алгоритм A\*

1. Теоретическая подготовка
   1. Алгоритмы поиска пути в графе:

* Алгоритм BFS: исследует граф, начиная с корневого узла и просматривая все соседние узлы на текущем уровне, прежде чем переходить к узлам на следующем уровне. Этот алгоритм использует очередь для хранения узлов, которые нужно исследовать.
* Алгоритм DFS: исследует граф, начиная с корневого узла и продвигаясь как можно глубже по одной ветви, прежде чем возвращаться назад. Этот алгоритм использует стек для хранения узлов, которые нужно исследовать.
* Алгоритм Dijkstra: используется для нахождения кратчайшего пути от одного узла до всех остальных узлов в взвешенном графе с неотрицательными весами рёбер. Он использует приоритетную очередь для хранения узлов, которые нужно исследовать.
* Алгоритм A\*: является расширением алгоритма Дейкстры и используется для нахождения кратчайшего пути в взвешенном графе с помощью эвристической функции. Он сочетает в себе стоимость пути от начального узла до текущего узла и оценку стоимости пути от текущего узла до целевого узла (*f*(*n*)=*g*(*n*)+*h*(*n*), где g(n) - фактическая стоимость пути от начального узла до узла n, h(n) - эвристическая функция, которая оценивает стоимость пути от узла n до целевого узла и основана на различных метриках, таких как евклидово расстояние или манхэттенское расстояние, f**(**n) - общая функция оценки, которая комбинирует g(n) и h(n) и используется для определения приоритетов при исследовании узлов). Он использует приоритетную очередь для хранения узлов, которые нужно исследовать.
  1. Вектор:
* Использование std::vector для динамического хранения элементов массива.
  1. Xеш-таблица:
* Использование std::unordered\_map для хранения пар ключ-значение с быстрым доступом по ключу.
  1. Стек:
* Использование std::stack для хранения элементов с доступом к последнему добавленному элементу.
  1. Очередь:
* Использование std::queue для хранения элементов с доступом к первому добавленному элементу.
* Использование std::priority\_queue для хранения элементов с доступом к первому элементу (со значением меньшего веса в данном случае)
  1. Числовые данные:
* int: используется для индексов массивов и счётчиков элементов, что позволяет эффективно управлять позициями, а также для представления значений элементов
* double: используется для хранения нецелочисленных данных

Для работы с графом я задала две структуры: Node и Graph, а также хэш-функцию для структуры Node (для работы с unordered\_map<Node>) (подробное описание в комментариях в коде ниже)

// Структура Node для представления узла графа  
struct Node {  
 double lon; // Долгота узла  
 double lat; // Широта узла  
  
 // Оператор сравнения для узлов  
 bool operator==(const Node other) const {  
 return this->lon == other.lon && this->lat == other.lat; // Проверяем равенство двух узлов по координатам  
 }  
  
 // Оператор неравенства для узлов  
 bool operator!=(const Node other) const {  
 return !(\*this == other); // Проверяем неравенство, используя оператор равенства  
 }  
  
 // Оператор меньше для узлов (для использования в приоритетной очереди)  
 bool operator<(const Node other) const {  
 if (lon != other.lon) // Сравниваем долготы  
 return lon < other.lon;  
 return lat < other.lat; // Если долготы равны, сравниваем широты  
 }  
};  
  
// Специализация хеш-функции для структуры Node (для работы с unordered\_map<Node>)  
namespace std {  
 template <>  
 struct hash<Node> {  
 // Определяем хеш-функцию для узлов, используя их координаты  
 size\_t operator()(const Node& node) const {  
 // Используем XOR для комбинирования хешей долготы и широты  
 return hash<double>()(node.lon) ^ hash<double>()(node.lat);  
 }  
 };  
}  
  
  
// Структура Graph для представления графа  
struct Graph {  
 // Список смежности графа, где ключ - узел, а значение - вектор пар (соседний узел, вес ребра)  
 unordered\_map<Node, vector<pair<Node, double>>> adjList;  
  
 // Метод для поиска ближайшего узла  
 Node find\_closest\_node(Node target) {  
 Node closest\_node = {0.0, 0.0}; // Инициализируем ближайший узел  
 double min\_distance = numeric\_limits<double>::infinity(); // Инициализируем минимальное расстояние  
  
 // Проходим по всем узлам в графе  
 for (auto pair: adjList) {  
 Node node = pair.first; // Получаем узел из списка смежности  
 double distance = sqrt(  
 pow(node.lon - target.lon, 2) + pow(node.lat - target.lat, 2)); // Вычисляем евклидово расстояние  
  
 if (distance < min\_distance) { // Если найдено более близкое расстояние  
 min\_distance = distance; // Обновляем минимальное расстояние  
 closest\_node = node; // Обновляем ближайший узел  
 }  
 }  
  
 return closest\_node; // Возвращаем ближайший узел  
 }  
  
 // Метод для парсинга данных графа из строки  
 void parseData(string data) {  
 stringstream ss(data); // Создаем строковый поток для разбора данных  
 string line;  
 while (getline(ss, line)) { // Читаем строки из потока  
 stringstream lineStream(line); // Создаем поток для текущей строки  
 double lon1, lat1, lon2, lat2, weight;  
 char comma;  
 // Читаем долготу и широту первого узла, ожидаем запятую и двоеточие  
 if (lineStream >> lon1 >> comma >> lat1 && comma == ',' && lineStream.get() == ':') {  
 Node node1{lon1, lat1}; // Создаем узел 1  
 // Читаем соседние узлы и веса ребер  
 while (lineStream >> lon2 >> comma >> lat2 >> comma >> weight) {  
 Node node2{lon2, lat2}; // Создаем узел 2  
 // Добавляем ребро в список смежности для обоих узлов  
 adjList[node1].emplace\_back(node2, weight);  
 adjList[node2].emplace\_back(node1, weight);  
 // Проверяем, есть ли конец списка соседей (символ ';')  
 if (!(lineStream >> comma) || comma != ';') {  
 break; // Выходим из цикла, если нет  
 }  
 }  
 }  
 }  
 }  
  
 // Метод для очистки графа  
 void clear() {  
 adjList.clear(); // Очищаем список смежности  
 }

1. Реализация алгоритма BFS

3.1. Реализованный метод

// V - кол-во вершин, E - кол-во рёбер  
// Метод для выполнения поиска в ширину (BFS)  
pair<pair<double, int>, vector<Node>> bfs(Node start, Node goal) {  
 // Очередь для хранения вершин и их текущих расстояний  
 queue<pair<Node, double>> q; // O(1) - создание очереди, пространственная сложность - O(V) - в худшем случае все вершины могут быть добавлены в очередь  
  
 // Хеш-таблицы для отслеживания посещенных вершин, расстояний и родительских вершин  
 unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
 unordered\_map<Node, double> distance; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
 unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
  
 // Начальная установка: добавляем начальную вершину в очередь с расстоянием 0  
 q.push({start, 0.0}); // O(1) - добавление в очередь  
 visited[start] = true; // O(1) - отмечаем начальную вершину как посещенную  
 distance[start] = 0.0; // O(1) - устанавливаем расстояние до начальной вершины как 0  
 parent[start] = start; // O(1) - родитель начальной вершины - сама она  
  
 // Основной цикл BFS  
 while (!q.empty()) { // O(V + E) - в худшем случае, когда все вершины и ребра будут обработаны  
 // Извлекаем текущую вершину и её расстояние из очереди  
 Node current = q.front().first; // O(1) - получение текущего узла, пространственная сложность - O(1)  
 double dist = q.front().second; // O(1) - получение расстояния, пространственная сложность - O(1)  
 q.pop(); // O(1) - удаление из очереди  
  
 // Если текущая вершина совпадает с целевой вершиной, восстанавливаем путь  
 if (current == goal) { // O(1) - проверка на совпадение  
 vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути  
 int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)  
  
 // Восстановление пути от goal до start  
 for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути  
 path.push\_back(at); // O(1) - добавляем узел в путь  
 edgeCount++; // O(1) - увеличиваем счетчик  
 }  
 path.push\_back(start); // O(1) - добавляем начальную вершину в путь  
  
 // Обращение пути, чтобы он начинался с start и заканчивался goal  
 reverse(path.begin(), path.end()); // O(P) - обращение вектора  
  
 // Возвращаем результат: расстояние, количество ребер и путь  
 return {{dist, edgeCount}, path}; // O(1) - возвращение результата  
 }  
  
 // Исследование соседних вершин текущей вершины  
 for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае  
 Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)  
 double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)  
  
 // Если соседняя вершина не была посещена  
 if (!visited[neighbor]) { // O(1) - проверка на посещенность  
 visited[neighbor] = true; // O(1) - отмечаем соседнюю вершину как посещенную  
 distance[neighbor] = dist + weight; // O(1) - обновляем расстояние до соседа  
 parent[neighbor] = current; // O(1) - устанавливаем родителя  
 q.push({neighbor, dist + weight}); // O(1) - добавляем соседа в очередь  
 }  
 }  
 }  
  
 // Если целевая вершина не была найдена, возвращаем бесконечное расстояние и пустой путь  
 return {{numeric\_limits<double>::infinity(), 0},  
 {}}; // O(1) - возвращение результата  
}  
// Общая пространственная сложность - O(V), общая временная сложность - O(V + E)

Метод для нахождения пути между вершинами с помощью алгоритма BFS, а также подсчета длины найденного пути с учетом весов ребер и без.

* 1. Основные действия
* **Инициализация структур данных**:

Создаются очередь q для хранения вершин и их текущих расстояний.

Создаются хеш-таблицы visited, distance и parent для отслеживания посещенных вершин, расстояний и родительских вершин соответственно.

* **Начальная установка:**

Начальная вершина start добавляется в очередь с расстоянием 0.

Вершина start помечается как посещенная, и её расстояние устанавливается в 0.

Родительской вершиной для start устанавливается сама start.

* **Основной цикл BFS**:

Пока очередь не пуста, извлекается вершина из начала очереди.

Если текущая вершина совпадает с целевой вершиной goal, восстанавливается путь от goal до start и возвращается результат.

* **Исследование соседних вершин**:

Для каждой соседней вершины текущей вершины проверяется, была ли она уже посещена.

Если соседняя вершина не была посещена, она помечается как посещенная, её расстояние устанавливается, и она добавляется в очередь с обновленным расстоянием.

* **Завершение алгоритма**:

Если очередь опустела, а целевая вершина не была найдена, возвращается результат с бесконечным расстоянием и пустым путем.

* 1. Временная и пространственная сложность

Общая пространственная сложность - O(V), общая временная сложность - O(V + E), где V – количество вершин, E – количество рёбер (подробный подсчёт в комментариях с кодом)

1. Реализация алгоритма DFS
   1. Реализованный метод

// Метод для выполнения поиска в глубину (DFS)  
pair<pair<double, int>, vector<Node>> dfs(Node start, Node goal) {  
 // Стек для хранения вершин и их текущих расстояний  
 stack<pair<Node, double>> s; // O(1) - создание стека, пространственная сложность - O(V) - в худшем случае все вершины могут быть добавлены в стек  
  
 // Хеш-таблицы для отслеживания посещенных вершин, расстояний и родительских вершин  
 unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
 unordered\_map<Node, double> distance; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
 unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)  
  
 // Начальная установка: добавляем начальную вершину в стек с расстоянием 0  
 s.push({start, 0.0}); // O(1) - добавление в стек  
 visited[start] = true; // O(1) - отмечаем начальную вершину как посещенную  
 distance[start] = 0.0; // O(1) - устанавливаем расстояние до начальной вершины как 0  
 parent[start] = start; // O(1) - родитель начальной вершины - сама она  
  
 // Основной цикл DFS  
 while (!s.empty()) { // O(V) - в худшем случае, когда все вершины будут обработаны  
 // Извлекаем текущую вершину и её расстояние из стека  
 Node current = s.top().first; // O(1) - получение текущего узла, пространственная сложность - O(1)  
 double dist = s.top().second; // O(1) - получение расстояния, пространственная сложность - O(1)  
 s.pop(); // O(1) - удаление из стека  
  
 // Если текущая вершина совпадает с целевой вершиной, восстанавливаем путь  
 if (current == goal) { // O(1) - проверка на совпадение  
 vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути  
 int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)  
  
 // Восстановление пути от goal до start  
 for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути  
 path.push\_back(at); // O(1) - добавляем узел в путь  
 edgeCount++; // O(1) - увеличиваем счетчик  
 }  
 path.push\_back(start); // O(1) - добавляем начальную вершину в путь  
  
 // Обращение пути, чтобы он начинался с start и заканчивался goal  
 reverse(path.begin(), path.end()); // O(P) - обращение вектора  
  
 // Возвращаем результат: расстояние, количество ребер и путь  
 return {{dist, edgeCount}, path}; // O(1) - возвращение результата  
 }  
  
 // Исследование соседних вершин текущей вершины  
 for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае  
 Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)  
 double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)  
  
 // Если соседняя вершина не была посещена  
 if (!visited[neighbor]) { // O(1) - проверка на посещенность  
 visited[neighbor] = true; // O(1) - отмечаем соседнюю вершину как посещенную  
 distance[neighbor] = dist + weight; // O(1) - обновляем расстояние до соседа  
 parent[neighbor] = current; // O(1) - устанавливаем родителя  
 s.push({neighbor, dist + weight}); // O(1) - добавляем соседа в стек  
 }  
 }  
 }  
  
 // Если целевая вершина не была найдена, возвращаем бесконечное расстояние и пустой путь  
 return {{numeric\_limits<double>::infinity(), 0},  
 {}}; // O(1) - возвращение результата  
}  
// Общая пространственная сложность - O(V), общая временная сложность - O(V + E)

Метод для нахождения пути между вершинами с помощью алгоритма DFS, а также подсчета длины найденного пути с учетом весов ребер и без.

* 1. Основные действия
  + Инициализация структур данных:

Создаются стек s для хранения вершин и их текущих расстояний.

Создаются хеш-таблицы visited, distance и parent для отслеживания посещенных вершин, расстояний и родительских вершин соответственно.

* + Начальная установка:

Начальная вершина start добавляется в стек с расстоянием 0.

Вершина start помечается как посещенная, и её расстояние устанавливается в 0.

Родительской вершиной для start устанавливается сама start.

* + Основной цикл DFS:

Пока стек не пуст, извлекается вершина из вершины стека.

Если текущая вершина совпадает с целевой вершиной goal, восстанавливается путь от goal до start и возвращается результат.

* + Исследование соседних вершин:

Для каждой соседней вершины текущей вершины проверяется, была ли она уже посещена.

Если соседняя вершина не была посещена, она помечается как посещенная, её расстояние устанавливается, и она добавляется в стек с обновленным расстоянием.

* + Завершение алгоритма:

Если стек опустел, а целевая вершина не была найдена, возвращается результат с бесконечным расстоянием и пустым путем.

* 1. Временная и пространственная сложность

Общая пространственная сложность - O(V), общая временная сложность - O(V + E), где V – количество вершин, E – количество рёбер (подробный подсчёт в комментариях с кодом)

1. Реализация алгоритма Dijkstra
   1. Реализованный метод

// Метод для выполнения алгоритма Дейкстры  
pair<pair<double, int>, vector<Node>> dijkstra(Node start, Node goal) {  
 // Приоритетная очередь для хранения узлов и их текущих расстояний (первыми в ней будут элементы с меньшим весом)  
 priority\_queue<pair<double, Node>, vector<pair<double, Node>>, greater<pair<double, Node>>> pq; // O(1) - создание приоритетной очереди, пространственная сложность - O(V) - в худшем случае все узлы могут быть добавлены в очередь  
  
 // Хеш-таблицы для хранения расстояний, посещенных узлов и родительских узлов  
 unordered\_map<Node, double> distances; // O(1) - создание хеш-таблицы для расстояний, пространственная сложность - O(V)  
 unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы для посещенных узлов, пространственная сложность - O(V)  
 unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы для родительских узлов, пространственная сложность - O(V)  
  
 // Инициализация расстояний до всех узлов как бесконечность  
 for (auto pair: adjList) { // O(V) - инициализация расстояний  
 distances[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка расстояния  
 }  
  
 // Установка начального расстояния до стартового узла  
 distances[start] = 0.0; // O(1) - установка расстояния до стартового узла  
 pq.push({0.0, start}); // O(log V) - добавление стартового узла в приоритетную очередь  
 parent[start] = start; // O(1) - родитель стартового узла - сама она  
  
 // Основной цикл алгоритма Дейкстры  
 while (!pq.empty()) { // O(V) - в худшем случае, когда все узлы будут обработаны  
 // Извлечение узла с минимальным расстоянием  
 double currentDistance = pq.top().first; // O(1) - получение минимального расстояния, пространственная сложность - O(1)  
 Node current = pq.top().second; // O(1) - получение узла, пространственная сложность - O(1)  
 pq.pop(); // O(log V) - удаление узла из приоритетной очереди  
  
 // Если текущий узел совпадает с целевым узлом, восстанавливаем путь  
 if (current == goal) { // O(1) - проверка на совпадение  
 vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути  
 int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)  
 for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути  
 path.push\_back(at); // O(1) - добавляем узел в путь  
 edgeCount++; // O(1) - увеличиваем счетчик  
 }  
 path.push\_back(start); // O(1) - добавляем начальную вершину в путь  
 reverse(path.begin(), path.end()); // O(P) - обращение пути  
 return {{currentDistance, edgeCount}, path}; // O(1) - возвращение результата  
 }  
  
 // Если узел уже посещен, пропускаем его  
 if (visited[current]) continue; // O(1) - проверка на посещенность  
 visited[current] = true; // O(1) - отмечаем текущий узел как посещенный  
  
 // Исследование соседних узлов  
 for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае, когда все соседи текущего узла будут обработаны  
 Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)  
 double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)  
  
 // Вычисление нового расстояния до соседнего узла  
 double newDist = currentDistance + weight; // O(1) - вычисление нового расстояния, пространственная сложность - O(1)  
  
 // Если новое расстояние меньше текущего расстояния до соседнего узла  
 if (newDist < distances[neighbor]) { // O(1) - проверка условия  
 distances[neighbor] = newDist; // O(1) - обновляем расстояние до соседа  
 parent[neighbor] = current; // O(1) - устанавливаем родителя  
 pq.push({newDist, neighbor}); // O(log V) - добавляем соседа в очередь  
 }  
 }  
 }  
  
 // Если целевой узел не был найден, возвращаем бесконечное расстояние и пустой путь  
 return {{numeric\_limits<double>::infinity(), 0},  
 {}}; // O(1) - возвращение результата  
}  
// Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V)

Метод для нахождения наименьшего пути между вершинами с помощью алгоритма Dijkstra, а также подсчета длины найденного пути с учетом весов ребер и без.

* 1. Основные действия
* Инициализация структур данных:

Создается приоритетная очередь pq для хранения узлов и их текущих расстояний.

Создаются хеш-таблицы distances, visited и parent для хранения расстояний, посещенных узлов и родительских узлов соответственно.

* Начальная установка:

Расстояния до всех узлов инициализируются как бесконечность.

Расстояние до начального узла start устанавливается в 0.

Начальный узел добавляется в приоритетную очередь с расстоянием 0.

Родительским узлом для start устанавливается сам start.

* Основной цикл алгоритма Дейкстры:

Пока приоритетная очередь не пуста, извлекается узел с минимальным расстоянием.

Если текущий узел совпадает с целевым узлом goal, восстанавливается путь от goal до start и возвращается результат.

* Исследование соседних узлов:

Для каждого соседнего узла текущего узла вычисляется новое расстояние.

Если новое расстояние меньше текущего расстояния до соседнего узла, обновляются расстояние и родительский узел для соседнего узла, и он добавляется в приоритетную очередь с обновленным расстоянием.

* Завершение алгоритма:

Если приоритетная очередь опустела, а целевой узел не был найден, возвращается результат с бесконечным расстоянием и пустым путем.

* 1. Временная и пространственная сложность

Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V), где V – количество вершин, E – количество рёбер (подробный подсчёт в комментариях с кодом)

1. Реализация алгоритма A\*
   1. Реализованный метод (+ вспомогательная эвристическая функция)

// Эвристическая функция для оценки расстояния между двумя узлами (евклидово расстояние)  
 double heuristic(Node a, Node b) {  
 return sqrt(pow(a.lon - b.lon, 2) + pow(a.lat - b.lat, 2)); // O(1) - вычисляем евклидово расстояние, пространственная сложность - O(1)  
 }  
  
 // Метод для выполнения алгоритма A\*  
 pair<pair<double, int>, vector<Node>> aStar(Node start, Node goal) {  
 // Приоритетная очередь для хранения узлов и их текущих fScore (первыми в ней будут элементы с меньшим весом)  
 priority\_queue<pair<double, Node>, vector<pair<double, Node>>, greater<pair<double, Node>>> pq; // O(1) - создание приоритетной очереди, пространственная сложность - O(V) - в худшем случае все узлы могут быть добавлены в очередь  
  
 // Хеш-таблицы для хранения gScore, fScore и родительских узлов  
 unordered\_map<Node, double> gScore; // O(1) - создание хеш-таблицы для gScore, пространственная сложность - O(V)  
 unordered\_map<Node, double> fScore; // O(1) - создание хеш-таблицы для fScore, пространственная сложность - O(V)  
 unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы для родительских узлов, пространственная сложность - O(V)  
  
 // Инициализация gScore и fScore для всех узлов как бесконечность  
 for (auto pair: adjList) { // O(V) - инициализация gScore и fScore  
 gScore[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка gScore  
 fScore[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка fScore  
 }  
  
 // Установка начального gScore и fScore для стартового узла  
 gScore[start] = 0.0; // O(1) - gScore для стартового узла  
 fScore[start] = heuristic(start, goal); // O(1) - fScore для стартового узла  
 parent[start] = start; // O(1) - родитель стартового узла - сама она  
  
 // Добавление стартового узла в приоритетную очередь  
 pq.push({fScore[start], start}); // O(log V) - добавление стартового узла в очередь  
  
 // Основной цикл алгоритма A\*  
 while (!pq.empty()) { // O(V) - в худшем случае, когда все узлы будут обработаны  
 // Извлечение узла с минимальным fScore  
 double currentFScore = pq.top().first; // O(1) - получение минимального fScore, пространственная сложность - O(1)  
 Node current = pq.top().second; // O(1) - получение узла, пространственная сложность - O(1)  
 pq.pop(); // O(log V) - удаление узла из приоритетной очереди  
  
 // Если текущий узел совпадает с целевым узлом, восстанавливаем путь  
 if (current == goal) { // O(1) - проверка на совпадение  
 vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути  
 int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)  
 for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути  
 path.push\_back(at); // O(1) - добавляем узел в путь  
 edgeCount++; // O(1) - увеличиваем счетчик  
 }  
 path.push\_back(start); // O(1) - добавляем начальную вершину в путь  
 reverse(path.begin(), path.end()); // O(P) - обращаем путь  
 return {{gScore[current], edgeCount}, path}; // O(1) - возвращаем результат  
 }  
  
 // Исследование соседних узлов  
 for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае, когда все соседи текущего узла будут обработаны  
 Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)  
 double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)  
  
 // Вычисление временного gScore для соседнего узла  
 double tentativeGScore = gScore[current] + weight; // O(1) - вычисление временного gScore, пространственная сложность - O(1)  
  
 // Если временный gScore меньше текущего gScore для соседнего узла  
 if (tentativeGScore < gScore[neighbor]) { // O(1) - проверка условия  
 // Обновление gScore и fScore для соседнего узла  
 gScore[neighbor] = tentativeGScore; // O(1) - обновляем gScore  
 fScore[neighbor] = tentativeGScore + heuristic(neighbor, goal); // O(1) - обновляем fScore  
 parent[neighbor] = current; // O(1) - устанавливаем родителя  
 pq.push({fScore[neighbor], neighbor}); // O(log V) - добавляем соседа в очередь  
 }  
 }  
 }  
  
 // Если целевой узел не был найден, возвращаем бесконечное расстояние и пустой путь  
 return {{numeric\_limits<double>::infinity(), 0},  
 {}}; // O(1) - возвращение результата  
 }  
 // Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V)  
};

Метод для нахождения наименьшего пути между вершинами с помощью алгоритма A\*, а также подсчета длины найденного пути с учетом весов ребер и без.

* 1. Основные действия
* Инициализация структур данных:

Создается приоритетная очередь pq для хранения узлов и их текущих fScore.

Создаются хеш-таблицы gScore, fScore и parent для хранения gScore, fScore и родительских узлов соответственно.

* Начальная установка:

gScore и fScore для всех узлов инициализируются как бесконечность.

Начальный gScore для стартового узла start устанавливается в 0.

Начальный fScore для стартового узла start устанавливается в значение эвристической функции между start и goal.

Родительским узлом для start устанавливается сам start.

Начальный узел добавляется в приоритетную очередь с его fScore.

* Основной цикл алгоритма A\*:

Пока приоритетная очередь не пуста, извлекается узел с минимальным fScore.

Если текущий узел совпадает с целевым узлом goal, восстанавливается путь от goal до start и возвращается результат.

* Исследование соседних узлов:

Для каждого соседнего узла текущего узла вычисляется временный gScore.

Если временный gScore меньше текущего gScore для соседнего узла, обновляются gScore и fScore для соседнего узла, и он добавляется в приоритетную очередь с обновленным fScore.

* Завершение алгоритма:

Если приоритетная очередь опустела, а целевой узел не был найден, возвращается результат с бесконечным расстоянием и пустым путем.

* 1. Временная и пространственная сложность

Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V), где V – количество вершин, E – количество рёбер (подробный подсчёт в комментариях с кодом)

1. Время работы алгоритмов

Время работы алгоритмов представлено на рисунке 1.

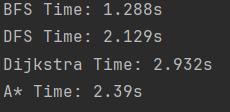


Рисунок 1 – Время работы алгоритмов

1. Сравнение методов и времени работы алгоритмов
   1. BFS
      * + Этот алгоритм не учитывает веса ребер при нахождении пути и ищет путь по количеству ребер в невзвешенном графе (находит оптимальный по количеству рёбер, совершая обход в ширину).
        + Время работы наименьшее среди всех алгоритмов, так как BFS игнорирует веса и находит путь по уровню. Подходит для обхода графов, нахождения кратчайшего пути между вершинами в невзвешенных графах.
   2. DFS
      * + Этот алгоритм не учитывает веса ребер при нахождении пути и ищет путь по количеству ребер в невзвешенном графе (не находит оптимальный путь, а просто обходит граф в глубину)
        + Время работы больше, чем у BFS, так как DFS совершает обход в глубину в графе без учета оптимальности пути. Его не стоит использовать для поиска кратчайшего пути, но его можно использовать для обхода.
   3. Dijkstra
      * + Время работы наибольшее среди всех алгоритмов.
        + Этот алгоритм учитывает веса ребер и гарантированно находит кратчайший путь во взвешенном графе с неотрицательными весами. Этот алгоритм медленнее остальных, поскольку учитывает веса и находит самый короткий путь. Это наиболее точный метод для поиска кратчайшего пути в графах с взвешенными ребрами.
   4. A\*
      * + A\* также учитывает веса ребер и использует эвристическую функцию для оценки стоимости пути до целевого узла, что позволяет ему находить оптимальный путь быстрее.
        + A\* показал время выполнения меньшее, чем Дейкстра, так как он сочетает в себе преимущества алгоритма Дейкстры и эвристической оценки, что позволяет ему более эффективно исследовать граф, но большее, чем BFS и DFS, так как учитывает веса рёбер. Его стоит использовать для задач, где требуется нахождение кратчайшего пути с предположениями о направлении поиска.
2. Выводы:
   * + - Алгоритмы Дейкстры и A\* находят оптимальный путь с минимальным расстоянием. BFS и DFS не находят оптимальный путь во взвешенных графах, так как они не учитывают веса ребер.
       - BFS позволяет совершить обход графа в ширину и найти минимальный путь между вершинами в невзвешенном графе. BFS не находит оптимальный путь во взвешенных графах, так как он не учитывает веса ребер.
       - DFS позволяет совершить обход графа в глубину, но не подходит для нахождения оптимального пути.
       - Дейкстра— самый точный для взвешенных графов, но уступает в скорости А\*.
       - A\* показывает время выполнения меньше, чем Дейкстра. Сбалансированный вариант: быстрее Дейкстры, но требует выбора правильной эвристики. Для больших графов A\* будет предпочтительнее Дейкстры из-за скорости, если есть хорошая эвристика.

ПРИЛОЖЕНИЕ А

Листинг кода min\_path.cpp

#include <iostream> // Подключение библиотеки для ввода-вывода

#include <unordered\_map> // Подключение библиотеки для работы с хеш-таблицами

#include <vector> // Подключение библиотеки для работы с динамическими массивами

#include <stack> // Подключение библиотеки для работы со стеком (для DFS)

#include <queue> // Подключение библиотеки для работы с очередью и приоритетной очередью (для BFS, Дейкстры и А\*)

#include <cmath> // Подключение библиотеки для математических функций (sqrt)

#include <fstream> // Подключение библиотеки для работы с файлами

#include <sstream> // Подключение библиотеки для работы со строковыми потоками

#include <chrono> // Подключение библиотеки для работы со временем

#include <cassert> // Подключение библиотеки для запуска тестов

#include <algorithm> // Подключение библиотеки для работы с алгоритмами (для использования reverse)

using namespace std; // Использование стандартного пространства имен для упрощения записи

// Структура Node для представления узла графа

struct Node {

double lon; // Долгота узла

double lat; // Широта узла

// Оператор сравнения для узлов

bool operator==(const Node other) const {

return this->lon == other.lon && this->lat == other.lat; // Проверяем равенство двух узлов по координатам

}

// Оператор неравенства для узлов

bool operator!=(const Node other) const {

return !(\*this == other); // Проверяем неравенство, используя оператор равенства

}

// Оператор меньше для узлов (для использования в приоритетной очереди)

bool operator<(const Node other) const {

if (lon != other.lon) // Сравниваем долготы

return lon < other.lon;

return lat < other.lat; // Если долготы равны, сравниваем широты

}

};

// Специализация хеш-функции для структуры Node (для работы с unordered\_map<Node>)

namespace std {

template <>

struct hash<Node> {

// Определяем хеш-функцию для узлов, используя их координаты

size\_t operator()(const Node& node) const {

// Используем XOR для комбинирования хешей долготы и широты

return hash<double>()(node.lon) ^ hash<double>()(node.lat);

}

};

}

// Структура Graph для представления графа

struct Graph {

// Список смежности графа, где ключ - узел, а значение - вектор пар (соседний узел, вес ребра)

unordered\_map<Node, vector<pair<Node, double>>> adjList;

// Метод для поиска ближайшего узла

Node find\_closest\_node(Node target) {

Node closest\_node = {0.0, 0.0}; // Инициализируем ближайший узел

double min\_distance = numeric\_limits<double>::infinity(); // Инициализируем минимальное расстояние

// Проходим по всем узлам в графе

for (auto pair: adjList) {

Node node = pair.first; // Получаем узел из списка смежности

double distance = sqrt(

pow(node.lon - target.lon, 2) + pow(node.lat - target.lat, 2)); // Вычисляем евклидово расстояние

if (distance < min\_distance) { // Если найдено более близкое расстояние

min\_distance = distance; // Обновляем минимальное расстояние

closest\_node = node; // Обновляем ближайший узел

}

}

return closest\_node; // Возвращаем ближайший узел

}

// Метод для парсинга данных графа из строки

void parseData(string data) {

stringstream ss(data); // Создаем строковый поток для разбора данных

string line;

while (getline(ss, line)) { // Читаем строки из потока

stringstream lineStream(line); // Создаем поток для текущей строки

double lon1, lat1, lon2, lat2, weight;

char comma;

// Читаем долготу и широту первого узла, ожидаем запятую и двоеточие

if (lineStream >> lon1 >> comma >> lat1 && comma == ',' && lineStream.get() == ':') {

Node node1{lon1, lat1}; // Создаем узел 1

// Читаем соседние узлы и веса ребер

while (lineStream >> lon2 >> comma >> lat2 >> comma >> weight) {

Node node2{lon2, lat2}; // Создаем узел 2

// Добавляем ребро в список смежности для обоих узлов

adjList[node1].emplace\_back(node2, weight);

adjList[node2].emplace\_back(node1, weight);

// Проверяем, есть ли конец списка соседей (символ ';')

if (!(lineStream >> comma) || comma != ';') {

break; // Выходим из цикла, если нет

}

}

}

}

}

// Метод для очистки графа

void clear() {

adjList.clear(); // Очищаем список смежности

}

// V - кол-во вершин, E - кол-во рёбер

// Метод для выполнения поиска в ширину (BFS)

pair<pair<double, int>, vector<Node>> bfs(Node start, Node goal) {

// Очередь для хранения вершин и их текущих расстояний

queue<pair<Node, double>> q; // O(1) - создание очереди, пространственная сложность - O(V) - в худшем случае все вершины могут быть добавлены в очередь

// Хеш-таблицы для отслеживания посещенных вершин, расстояний и родительских вершин

unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

unordered\_map<Node, double> distance; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

// Начальная установка: добавляем начальную вершину в очередь с расстоянием 0

q.push({start, 0.0}); // O(1) - добавление в очередь

visited[start] = true; // O(1) - отмечаем начальную вершину как посещенную

distance[start] = 0.0; // O(1) - устанавливаем расстояние до начальной вершины как 0

parent[start] = start; // O(1) - родитель начальной вершины - сама она

// Основной цикл BFS

while (!q.empty()) { // O(V + E) - в худшем случае, когда все вершины и ребра будут обработаны

// Извлекаем текущую вершину и её расстояние из очереди

Node current = q.front().first; // O(1) - получение текущего узла, пространственная сложность - O(1)

double dist = q.front().second; // O(1) - получение расстояния, пространственная сложность - O(1)

q.pop(); // O(1) - удаление из очереди

// Если текущая вершина совпадает с целевой вершиной, восстанавливаем путь

if (current == goal) { // O(1) - проверка на совпадение

vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути

int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)

// Восстановление пути от goal до start

for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути

path.push\_back(at); // O(1) - добавляем узел в путь

edgeCount++; // O(1) - увеличиваем счетчик

}

path.push\_back(start); // O(1) - добавляем начальную вершину в путь

// Обращение пути, чтобы он начинался с start и заканчивался goal

reverse(path.begin(), path.end()); // O(P) - обращение вектора

// Возвращаем результат: расстояние, количество ребер и путь

return {{dist, edgeCount}, path}; // O(1) - возвращение результата

}

// Исследование соседних вершин текущей вершины

for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае

Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)

double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)

// Если соседняя вершина не была посещена

if (!visited[neighbor]) { // O(1) - проверка на посещенность

visited[neighbor] = true; // O(1) - отмечаем соседнюю вершину как посещенную

distance[neighbor] = dist + weight; // O(1) - обновляем расстояние до соседа

parent[neighbor] = current; // O(1) - устанавливаем родителя

q.push({neighbor, dist + weight}); // O(1) - добавляем соседа в очередь

}

}

}

// Если целевая вершина не была найдена, возвращаем бесконечное расстояние и пустой путь

return {{numeric\_limits<double>::infinity(), 0},

{}}; // O(1) - возвращение результата

}

// Общая пространственная сложность - O(V), общая временная сложность - O(V + E)

// Метод для выполнения поиска в глубину (DFS)

pair<pair<double, int>, vector<Node>> dfs(Node start, Node goal) {

// Стек для хранения вершин и их текущих расстояний

stack<pair<Node, double>> s; // O(1) - создание стека, пространственная сложность - O(V) - в худшем случае все вершины могут быть добавлены в стек

// Хеш-таблицы для отслеживания посещенных вершин, расстояний и родительских вершин

unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

unordered\_map<Node, double> distance; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы, пространственная сложность - O(V)

// Начальная установка: добавляем начальную вершину в стек с расстоянием 0

s.push({start, 0.0}); // O(1) - добавление в стек

visited[start] = true; // O(1) - отмечаем начальную вершину как посещенную

distance[start] = 0.0; // O(1) - устанавливаем расстояние до начальной вершины как 0

parent[start] = start; // O(1) - родитель начальной вершины - сама она

// Основной цикл DFS

while (!s.empty()) { // O(V) - в худшем случае, когда все вершины будут обработаны

// Извлекаем текущую вершину и её расстояние из стека

Node current = s.top().first; // O(1) - получение текущего узла, пространственная сложность - O(1)

double dist = s.top().second; // O(1) - получение расстояния, пространственная сложность - O(1)

s.pop(); // O(1) - удаление из стека

// Если текущая вершина совпадает с целевой вершиной, восстанавливаем путь

if (current == goal) { // O(1) - проверка на совпадение

vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути

int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)

// Восстановление пути от goal до start

for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути

path.push\_back(at); // O(1) - добавляем узел в путь

edgeCount++; // O(1) - увеличиваем счетчик

}

path.push\_back(start); // O(1) - добавляем начальную вершину в путь

// Обращение пути, чтобы он начинался с start и заканчивался goal

reverse(path.begin(), path.end()); // O(P) - обращение вектора

// Возвращаем результат: расстояние, количество ребер и путь

return {{dist, edgeCount}, path}; // O(1) - возвращение результата

}

// Исследование соседних вершин текущей вершины

for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае

Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)

double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)

// Если соседняя вершина не была посещена

if (!visited[neighbor]) { // O(1) - проверка на посещенность

visited[neighbor] = true; // O(1) - отмечаем соседнюю вершину как посещенную

distance[neighbor] = dist + weight; // O(1) - обновляем расстояние до соседа

parent[neighbor] = current; // O(1) - устанавливаем родителя

s.push({neighbor, dist + weight}); // O(1) - добавляем соседа в стек

}

}

}

// Если целевая вершина не была найдена, возвращаем бесконечное расстояние и пустой путь

return {{numeric\_limits<double>::infinity(), 0},

{}}; // O(1) - возвращение результата

}

// Общая пространственная сложность - O(V), общая временная сложность - O(V + E)

// Метод для выполнения алгоритма Дейкстры

pair<pair<double, int>, vector<Node>> dijkstra(Node start, Node goal) {

// Приоритетная очередь для хранения узлов и их текущих расстояний (первыми в ней будут элементы с меньшим весом)

priority\_queue<pair<double, Node>, vector<pair<double, Node>>, greater<pair<double, Node>>> pq; // O(1) - создание приоритетной очереди, пространственная сложность - O(V) - в худшем случае все узлы могут быть добавлены в очередь

// Хеш-таблицы для хранения расстояний, посещенных узлов и родительских узлов

unordered\_map<Node, double> distances; // O(1) - создание хеш-таблицы для расстояний, пространственная сложность - O(V)

unordered\_map<Node, bool> visited; // O(1) - создание хеш-таблицы для посещенных узлов, пространственная сложность - O(V)

unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы для родительских узлов, пространственная сложность - O(V)

// Инициализация расстояний до всех узлов как бесконечность

for (auto pair: adjList) { // O(V) - инициализация расстояний

distances[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка расстояния

}

// Установка начального расстояния до стартового узла

distances[start] = 0.0; // O(1) - установка расстояния до стартового узла

pq.push({0.0, start}); // O(log V) - добавление стартового узла в приоритетную очередь

parent[start] = start; // O(1) - родитель стартового узла - сама она

// Основной цикл алгоритма Дейкстры

while (!pq.empty()) { // O(V) - в худшем случае, когда все узлы будут обработаны

// Извлечение узла с минимальным расстоянием

double currentDistance = pq.top().first; // O(1) - получение минимального расстояния, пространственная сложность - O(1)

Node current = pq.top().second; // O(1) - получение узла, пространственная сложность - O(1)

pq.pop(); // O(log V) - удаление узла из приоритетной очереди

// Если текущий узел совпадает с целевым узлом, восстанавливаем путь

if (current == goal) { // O(1) - проверка на совпадение

vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути

int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)

for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути

path.push\_back(at); // O(1) - добавляем узел в путь

edgeCount++; // O(1) - увеличиваем счетчик

}

path.push\_back(start); // O(1) - добавляем начальную вершину в путь

reverse(path.begin(), path.end()); // O(P) - обращение пути

return {{currentDistance, edgeCount}, path}; // O(1) - возвращение результата

}

// Если узел уже посещен, пропускаем его

if (visited[current]) continue; // O(1) - проверка на посещенность

visited[current] = true; // O(1) - отмечаем текущий узел как посещенный

// Исследование соседних узлов

for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае, когда все соседи текущего узла будут обработаны

Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)

double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)

// Вычисление нового расстояния до соседнего узла

double newDist = currentDistance + weight; // O(1) - вычисление нового расстояния, пространственная сложность - O(1)

// Если новое расстояние меньше текущего расстояния до соседнего узла

if (newDist < distances[neighbor]) { // O(1) - проверка условия

distances[neighbor] = newDist; // O(1) - обновляем расстояние до соседа

parent[neighbor] = current; // O(1) - устанавливаем родителя

pq.push({newDist, neighbor}); // O(log V) - добавляем соседа в очередь

}

}

}

// Если целевой узел не был найден, возвращаем бесконечное расстояние и пустой путь

return {{numeric\_limits<double>::infinity(), 0},

{}}; // O(1) - возвращение результата

}

// Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V)

// Эвристическая функция для оценки расстояния между двумя узлами (евклидово расстояние)

double heuristic(Node a, Node b) {

return sqrt(pow(a.lon - b.lon, 2) + pow(a.lat - b.lat, 2)); // O(1) - вычисляем евклидово расстояние, пространственная сложность - O(1)

}

// Метод для выполнения алгоритма A\*

pair<pair<double, int>, vector<Node>> aStar(Node start, Node goal) {

// Приоритетная очередь для хранения узлов и их текущих fScore (первыми в ней будут элементы с меньшим весом)

priority\_queue<pair<double, Node>, vector<pair<double, Node>>, greater<pair<double, Node>>> pq; // O(1) - создание приоритетной очереди, пространственная сложность - O(V) - в худшем случае все узлы могут быть добавлены в очередь

// Хеш-таблицы для хранения gScore, fScore и родительских узлов

unordered\_map<Node, double> gScore; // O(1) - создание хеш-таблицы для gScore, пространственная сложность - O(V)

unordered\_map<Node, double> fScore; // O(1) - создание хеш-таблицы для fScore, пространственная сложность - O(V)

unordered\_map<Node, Node> parent; // O(1) - создание хеш-таблицы для родительских узлов, пространственная сложность - O(V)

// Инициализация gScore и fScore для всех узлов как бесконечность

for (auto pair: adjList) { // O(V) - инициализация gScore и fScore

gScore[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка gScore

fScore[pair.first] = numeric\_limits<double>::infinity(); // O(1) - установка fScore

}

// Установка начального gScore и fScore для стартового узла

gScore[start] = 0.0; // O(1) - gScore для стартового узла

fScore[start] = heuristic(start, goal); // O(1) - fScore для стартового узла

parent[start] = start; // O(1) - родитель стартового узла - сама она

// Добавление стартового узла в приоритетную очередь

pq.push({fScore[start], start}); // O(log V) - добавление стартового узла в очередь

// Основной цикл алгоритма A\*

while (!pq.empty()) { // O(V) - в худшем случае, когда все узлы будут обработаны

// Извлечение узла с минимальным fScore

double currentFScore = pq.top().first; // O(1) - получение минимального fScore, пространственная сложность - O(1)

Node current = pq.top().second; // O(1) - получение узла, пространственная сложность - O(1)

pq.pop(); // O(log V) - удаление узла из приоритетной очереди

// Если текущий узел совпадает с целевым узлом, восстанавливаем путь

if (current == goal) { // O(1) - проверка на совпадение

vector<Node> path; // Вектор для хранения пути, пространственная сложность - O(P), где P — длина пути

int edgeCount = 1; // Счетчик ребер, пространственная сложность - O(1)

for (Node at = goal; at != start; at = parent[at]) { // O(P) - P - длина пути

path.push\_back(at); // O(1) - добавляем узел в путь

edgeCount++; // O(1) - увеличиваем счетчик

}

path.push\_back(start); // O(1) - добавляем начальную вершину в путь

reverse(path.begin(), path.end()); // O(P) - обращаем путь

return {{gScore[current], edgeCount}, path}; // O(1) - возвращаем результат

}

// Исследование соседних узлов

for (auto neighbor\_pair: adjList[current]) { // O(E) - в худшем случае, когда все соседи текущего узла будут обработаны

Node neighbor = neighbor\_pair.first; // O(1) - получение соседнего узла, пространственная сложность - O(1)

double weight = neighbor\_pair.second; // O(1) - получение веса ребра, пространственная сложность - O(1)

// Вычисление временного gScore для соседнего узла

double tentativeGScore = gScore[current] + weight; // O(1) - вычисление временного gScore, пространственная сложность - O(1)

// Если временный gScore меньше текущего gScore для соседнего узла

if (tentativeGScore < gScore[neighbor]) { // O(1) - проверка условия

// Обновление gScore и fScore для соседнего узла

gScore[neighbor] = tentativeGScore; // O(1) - обновляем gScore

fScore[neighbor] = tentativeGScore + heuristic(neighbor, goal); // O(1) - обновляем fScore

parent[neighbor] = current; // O(1) - устанавливаем родителя

pq.push({fScore[neighbor], neighbor}); // O(log V) - добавляем соседа в очередь

}

}

}

// Если целевой узел не был найден, возвращаем бесконечное расстояние и пустой путь

return {{numeric\_limits<double>::infinity(), 0},

{}}; // O(1) - возвращение результата

}

// Общая пространственная сложность - O(V), общая временная сложность - O((V + E) log V)

};

// Функция для запуска тестов

void runTests() {

Graph graph; // Создаем экземпляр графа

// Тест 1: Простой граф с прямым путем

string data1 = R"(

1.0,1.0:2.0,2.0,1.0;

2.0,2.0:3.0,3.0,1.0;

3.0,3.0:4.0,4.0,1.0;

)";

graph.parseData(data1); // Парсим данные графа из строки

Node start1{ 1.0, 1.0 }; // Определяем начальную вершину

Node goal1{ 4.0, 4.0 }; // Определяем целевую вершину

// Проверяем, что все алгоритмы возвращают правильное расстояние

assert(graph.bfs(start1, goal1).first.first == 3.0); // Проверка BFS

assert(graph.dfs(start1, goal1).first.first == 3.0); // Проверка DFS

assert(graph.dijkstra(start1, goal1).first.first == 3.0); // Проверка алгоритма Дейкстры

assert(graph.aStar(start1, goal1).first.first == 3.0); // Проверка алгоритма A\*

graph.clear(); // Очищаем граф для следующего теста

// Тест 2: Граф без пути

string data2 = R"(

1.0,1.0:2.0,2.0,1.0;

3.0,3.0:4.0,4.0,1.0;

)";

graph.parseData(data2); // Парсим данные графа из строки

Node start2{ 1.0, 1.0 }; // Определяем начальную вершину

Node goal2{ 4.0, 4.0 }; // Определяем целевую вершину

// Проверяем, что все алгоритмы возвращают бесконечное расстояние

assert(graph.bfs(start2, goal2).first.first == numeric\_limits<double>::infinity()); // Проверка BFS

assert(graph.dfs(start2, goal2).first.first == numeric\_limits<double>::infinity()); // Проверка DFS

assert(graph.dijkstra(start2, goal2).first.first == numeric\_limits<double>::infinity()); // Проверка алгоритма Дейкстры

assert(graph.aStar(start2, goal2).first.first == numeric\_limits<double>::infinity()); // Проверка алгоритма A\*

graph.clear(); // Очищаем граф для следующего теста

// Тест 3: Граф с циклом

string data3 = R"(

1.0,1.0:2.0,2.0,1.0;

2.0,2.0:3.0,3.0,1.0;

3.0,3.0:1.0,1.0,1.0;

)";

graph.parseData(data3); // Парсим данные графа из строки

Node start3{ 1.0, 1.0 }; // Определяем начальную вершину

Node goal3{ 3.0, 3.0 }; // Определяем целевую вершину

// Проверяем, что все алгоритмы возвращают правильное расстояние

assert(graph.bfs(start3, goal3).first.first == 1.0); // Проверка BFS

assert(graph.dfs(start3, goal3).first.first == 1.0); // Проверка DFS

assert(graph.dijkstra(start3, goal3).first.first == 1.0); // Проверка алгоритма Дейкстры

assert(graph.aStar(start3, goal3).first.first == 1.0); // Проверка алгоритма A\*

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

}

// Функция для тестирования данных из файла

void test\_file() {

Graph graph; // Создаем экземпляр графа

// Чтение данных из файла

ifstream inputFile("C:/Users/Lena/Downloads/graph.txt"); // Открываем файл для чтения

if (!inputFile.is\_open()) { // Проверяем, был ли файл успешно открыт

cerr << "Не удалось открыть файл!" << endl;

return;

}

stringstream buffer; // Создаем поток для хранения данных

buffer << inputFile.rdbuf(); // Считываем содержимое файла в поток

string data = buffer.str(); // Преобразуем поток в строку

graph.parseData(data); // Парсим данные графа

Node start{ 1.0, 1.0 }; // Определяем начальную вершину

Node goal{ 5.0, 5.0 }; // Определяем целевую вершину

// Проверяем, что все алгоритмы возвращают правильное расстояние

assert(graph.bfs(start, goal).first.first == 6.0); // Проверка BFS

assert(graph.dfs(start, goal).first.first == 6.0); // Проверка DFS

assert(graph.dijkstra(start, goal).first.first == 6.0); // Проверка алгоритма Дейкстры

assert(graph.aStar(start, goal).first.first == 6.0); // Проверка алгоритма A\*

// Запуск алгоритмов и замер времени

auto startBFS = chrono::high\_resolution\_clock::now(); // Начало замера времени для BFS

auto bfsResult = graph.bfs(start, goal); // Запуск BFS

auto endBFS = chrono::high\_resolution\_clock::now(); // Конец замера времени для BFS

cout << "BFS Distance: " << bfsResult.first.first << " Edges: " << bfsResult.first.second << " Time: "

<< chrono::duration\_cast<chrono::microseconds>(endBFS - startBFS).count() << "ms" << endl; // Выводим результаты BFS

cout << "BFS Path: "; // Выводим путь, найденный BFS

for (auto node : bfsResult.second) {

cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

}

cout << endl;

auto startDFS = chrono::high\_resolution\_clock::now(); // Начало замера времени для DFS

auto dfsResult = graph.dfs(start, goal); // Запуск DFS

auto endDFS = chrono::high\_resolution\_clock::now(); // Конец замера времени для DFS

cout << "DFS Distance: " << dfsResult.first.first << " Edges: " << dfsResult.first.second << " Time: "

<< chrono::duration\_cast<chrono::microseconds>(endDFS - startDFS).count() << "ms" << endl; // Выводим результаты DFS

cout << "DFS Path: "; // Выводим путь, найденный DFS

for (auto node : dfsResult.second) {

cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

}

cout << endl;

auto startDijkstra = chrono::high\_resolution\_clock::now(); // Начало замера времени для алгоритма Дейкстры

auto dijkstraResult = graph.dijkstra(start, goal); // Запуск алгоритма Дейкстры

auto endDijkstra = chrono::high\_resolution\_clock::now(); // Конец замера времени для алгоритма Дейкстры

cout << "Dijkstra Distance: " << dijkstraResult.first.first << " Edges: " << dijkstraResult.first.second << " Time: "

<< chrono::duration\_cast<chrono::microseconds>(endDijkstra - startDijkstra).count() << "ms" << endl; // Выводим результаты алгоритма Дейкстры

cout << "Dijkstra Path: "; // Выводим путь, найденный алгоритмом Дейкстры

for (auto node : dijkstraResult.second) {

cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

}

cout << endl;

auto startAStar = chrono::high\_resolution\_clock::now(); // Начало замера времени для алгоритма A\*

auto aStarResult = graph.aStar(start, goal); // Запуск алгоритма A\*

auto endAStar = chrono::high\_resolution\_clock::now(); // Конец замера времени для алгоритма A\*

cout << "A\* Distance: " << aStarResult.first.first << " Edges: " << aStarResult.first.second << " Time: "

<< chrono::duration\_cast<chrono::microseconds>(endAStar - startAStar).count() << "ms" << endl; // Выводим результаты алгоритма A\*

cout << "A\* Path: "; // Выводим путь, найденный алгоритмом A\*

for (auto node : aStarResult.second) {

cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

}

cout << endl;

cout << "Test\_file passed!" << endl;

}

int main() {

// Запускаем тесты, чтобы проверить корректность работы алгоритмов

runTests();

// Тестируем обработку данных из файла

test\_file();

Graph graph; // Создаем экземпляр графа для работы с данными

// Чтение данных из файла

ifstream inputFile("C:/Users/Lena/Downloads/spb\_graph (1).txt"); // Открываем файл для чтения данных графа

if (!inputFile.is\_open()) { // Проверяем, был ли файл успешно открыт

cerr << "Не удалось открыть файл!" << endl;

return 1;

}

stringstream buffer; // Создаем поток для хранения данных из файла

buffer << inputFile.rdbuf(); // Считываем содержимое файла в поток

string data = buffer.str(); // Преобразуем поток в строку, содержащую данные графа

graph.parseData(data); // Парсим данные графа из строки

// Определяем стартовую и целевую вершины для поиска пути

Node start{ 30.462547, 59.916606 }; // Стартовая вершина

Node goal{ 30.308108, 59.957238 }; // Целевая вершина

// Запуск алгоритмов поиска пути и замер времени выполнения

// Ниже приведены примеры замера времени и вывода результатов для различных алгоритмов.

// Запуск и замер времени для алгоритма BFS

// auto startBFS = chrono::high\_resolution\_clock::now(); // Начало замера времени для BFS

// auto bfsResult = graph.bfs(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запуск алгоритма BFS для поиска пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

// auto endBFS = chrono::high\_resolution\_clock::now(); // Конец замера времени для BFS

// Выводим результаты BFS и время выполнения в секундах

// cout << "BFS Distance: " << bfsResult.first.first << " Edges: " << bfsResult.first.second << " Time: "

// << chrono::duration\_cast<chrono::milliseconds>(endBFS - startBFS).count() / 1000.0 << "s" << endl;

// cout << "BFS Path: "; // Выводим путь, найденный BFS

// for (auto node : bfsResult.second) {

// cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

// }

// cout << endl;

// Аналогично для алгоритма DFS

// auto startDFS = chrono::high\_resolution\_clock::now(); // Начало замера времени для DFS

// auto dfsResult = graph.dfs(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запуск алгоритма DFS для поиска пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

// auto endDFS = chrono::high\_resolution\_clock::now(); // Конец замера времени для DFS

// Выводим результаты DFS и время выполнения в секундах

// cout << "DFS Distance: " << dfsResult.first.first << " Edges: " << dfsResult.first.second << " Time: "

// << chrono::duration\_cast<chrono::milliseconds>(endDFS - startDFS).count() / 1000.0 << "s" << endl;

// cout << "DFS Path: "; // Выводим путь, найденный DFS

// for (auto node : dfsResult.second) {

// cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

// }

// cout << endl;

// Аналогично для алгоритма Дейкстры

// auto startDijkstra = chrono::high\_resolution\_clock::now(); // Начало замера времени для алгоритма Дейкстры

// auto dijkstraResult = graph.dijkstra(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запуск алгоритма Дейкстры для поиска кратчайшего пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

// auto endDijkstra = chrono::high\_resolution\_clock::now(); // Конец замера времени для алгоритма Дейкстры

// Выводим результаты алгоритма Дейкстры и время выполнения в секундах

// cout << "Dijkstra Distance: " << dijkstraResult.first.first << " Edges: " << dijkstraResult.first.second << " Time: "

// << chrono::duration\_cast<chrono::milliseconds>(endDijkstra - startDijkstra).count() / 1000.0 << "s" << endl;

// cout << "Dijkstra Path: "; // Выводим путь, найденный алгоритмом Дейкстры

// for (auto node : dijkstraResult.second) {

// cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

// }

// cout << endl;

// Аналогично для алгоритма A\*

// auto startAStar = chrono::high\_resolution\_clock::now(); // Начало замера времени для алгоритма A\*

// auto aStarResult = graph.aStar(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запуск алгоритма A\* для поиска кратчайшего пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

// auto endAStar = chrono::high\_resolution\_clock::now(); // Конец замера времени для алгоритма A\*

// Выводим результаты алгоритма A\* и время выполнения в секундах

// cout << "A\* Distance: " << aStarResult.first.first << " Edges: " << aStarResult.first.second << " Time: "

// << chrono::duration\_cast<chrono::milliseconds>(endAStar - startAStar).count() / 1000.0 << "s" << endl;

// cout << "A\* Path: "; // Выводим путь, найденный алгоритмом A\*

// for (auto node : aStarResult.second) {

// cout << "(" << node.lon << ", " << node.lat << ") "; // Форматируем вывод координат узлов пути

// }

// cout << endl;

// Запуск алгоритмов поиска пути и замер времени выполнения каждого алгоритма

// Начало замера времени для алгоритма BFS (поиск в ширину)

auto startBFS = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время перед запуском алгоритма

auto bfsResult = graph.bfs(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запускаем алгоритм BFS для поиска пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

auto endBFS = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время после завершения алгоритма

// Выводим время выполнения алгоритма BFS в секундах

cout << "BFS Time: "

<< chrono::duration\_cast<chrono::milliseconds>(endBFS - startBFS).count() / 1000.0 << "s" << endl; // Преобразуем разницу времени в секунды и выводим

// Начало замера времени для алгоритма DFS (поиск в глубину)

auto startDFS = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время перед запуском алгоритма

auto dfsResult = graph.dfs(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запускаем алгоритм DFS для поиска пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

auto endDFS = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время после завершения алгоритма

// Выводим время выполнения алгоритма DFS в секундах

cout << "DFS Time: "

<< chrono::duration\_cast<chrono::milliseconds>(endDFS - startDFS).count() / 1000.0 << "s" << endl; // Преобразуем разницу времени в секунды и выводим

// Начало замера времени для алгоритма Дейкстры

auto startDijkstra = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время перед запуском алгоритма

auto dijkstraResult = graph.dijkstra(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запускаем алгоритм Дейкстры для поиска кратчайшего пути от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

auto endDijkstra = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время после завершения алгоритма

// Выводим время выполнения алгоритма Дейкстры в секундах

cout << "Dijkstra Time: "

<< chrono::duration\_cast<chrono::milliseconds>(endDijkstra - startDijkstra).count() / 1000.0 << "s" << endl; // Преобразуем разницу времени в секунды и выводим

// Начало замера времени для алгоритма A\*

auto startAStar = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время перед запуском алгоритма

auto aStarResult = graph.aStar(graph.find\_closest\_node(start), graph.find\_closest\_node(goal)); // Запускаем алгоритм A\* для поиска кратчайшего пути с учетом эвристики от ближайшей к стартовой вершины до ближайшей к целевой вершины из файла

auto endAStar = chrono::high\_resolution\_clock::now(); // Запоминаем текущее время после завершения алгоритма

// Выводим время выполнения алгоритма A\* в секундах

cout << "A\* Time: "

<< chrono::duration\_cast<chrono::milliseconds>(endAStar - startAStar).count() / 1000.0 << "s" << endl; // Преобразуем разницу времени в секунды и выводим

return 0;

}