ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе №8

«Графы, алгоритмы поиска кратчайшего пути»

Выполнил работу

Кащеев Максим

Академическая группа №J3113

Принято

Иван Владимирович Ходненко

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Найти путь алгоритмами обхода в ширину (BFS), в глубину (DFS), А\*, алгоритмом Дейкстры от места вашего постоянного проживания в Санкт-Петербурге к университету ИТМО на переданном графе.

1. Теоретическая подготовка

Для выполнения данной лабораторной работы необходимо знать алгоритмы обхода графов BFS, DFS, A\* и алгоритм Дейкстры. Также необходимо знать библиотеки и структуры данных, такие как iostream, fstream, vector, queue, stack, unordered\_set, unordered\_map, pair, string, limits. Необходимо уметь работать с указателями, вводом и выводом.

1. Реализация

В ходе выполнения данной лабораторной работы я реализовывал алгоритмы обхода графа для поиска кратчайшего пути в одном файле, в виде методов структуры Graph. Также для загрузки графа из файла создан метод load\_graph\_from\_file который читает исходные данные графа из файла, создает соответствующие узлы и возвращает экземпляр структуры Graph.

Для работы алгоритмов созданы следующие структуры и глобальные переменные: Node – для хранения узлов графа, идентификатора, координат, и списка ссылок на дочерние узлы; CompareNode – структура компаратор для использования в ordered структурах внутри алгоритма, работающих с указателями; splitString – функция разделения строки по символам, используется при начальной загрузке данных, при парсинге строк; start и end – переменные для хранения временных меток для вывода времени работы алгоритмов; start\_counter, end\_counter – функции для отметки старта и завершения работы алгоритма, и для вывода полученного промежутка времени.

В структуре Graph также созданы дополнительные методы необходимые для работы реализованных алгоритмов: getOrAddNodeByCoordinatesLine – метод находящий запрошенный по исходной строке координат узел, или создающий его при отсутствии; findClosestNode – метод поиска узла по переданным координатам и расстоянию; distance\_between – метод эвристики для подсчета расстояния по прямой на карте, исходя из координат; reconstructPath – перерабатывает набор собранных алгоритмом узлов в вектор идентификаторов; getGCost – метод возвращающий значение из переданного unordered map в случае если оно есть, либо бесконечность если его нет, используется для исключения инициализации полного массива весов для алгоритмов обхода.

Все алгоритмы поиска пути реализованы в виде методов структуры Graph, принимающих на вход два указателя на узлы для поиска пути, и возвращающие найденный путь и его длину. start\_counter и stop\_counter вызываются на старте и после завершения тела алгоритма, для замера времени выполнения алгоритма и его вывода. В теле каждого метода проверяется являются ли начальный и конечный узлы одним и тем же узлом, и тогда возвращается ответ из этого элемента в пути и нулевым расстоянием.

Для реализации поиска минимального пути в графе с использованием алгоритма BFS написан метод bfs\_shortest. После начальных проверок создаются переменные для обхода – queue – очередь для хранения указателей на те узлы графа, которые найдены, но у которых соседи еще не обработаны. came\_from – контейнер для накопления пути найденного в процессе обхода. distances – хеш таблица расстояний от start\_node до индексированного в хеше узла. visited – множество содержащее идентификаторы посещенных узлов. В процессе работы алгоритма производятся следующие действия – помешаем стартовый узел в очередь queue для старта обхода, и задаем расстояние как нулевое в distances, а также помечаем узел как посещенный в visited, затем начинаем обработку в цикле, где на каждой итерации достаем головной узел из очереди, в current\_node проверяем не является ли он искомым, и если да то переходим к сборке ответа и возвращаем его, если нет, то достаем соседей, current\_node->nodes и для каждого еще не посещенного соседа добавляем его в очередь, отмечаем в visited, выставляем расстояния в distances, запоминаем в came\_from откуда пришли в этот узел. Также если цикл завершился вследствие того что очередь пуста, то значит что путь не существует, выходим с пустым путем и бесконечным расстоянием.

Для реализации поиска пути в графе с использованием алгоритма DFS написан метод dfs. После начальных проверок создаются переменные для обхода – st – стек для обхода в глубину. visited – множество содержащее идентификаторы посещенных узлов. came\_from – контейнер для накопления пути найденного в процессе обхода. После начальной инициализации st начальным узлом, и отметке в visited этого узла начинается основной цикл, в котором обрабатывается текущий узел current из вершины стека, проверяется является ли он искомым, и если да производится сборка и возвращение результата, а если нет, то достаем соседей current->nodes и еще не посещенных обрабатываем соседей, добавляя их в стек, и отмечая как посещенные. Также если цикл завершился вследствие того что очередь пуста, то значит что путь не существует, выходим с пустым путем и бесконечным расстоянием.

Для реализации поиска кратчайшего пути в графе по алгоритму Дейкстры написан метод dijkstra\_shortest. После начальных проверок создаются переменные для обхода – pq – приоритетная очередь, для упорядоченного хранения узлов по расстоянию, чтобы автоматически получать ближайший узел на каждом шаге. distances – таблица расстояний от начального узла до других узлов. came\_from – контейнер для накопления пути найденного в процессе обхода. После начальной инициализации pq начальным узлом, и distances нулевым расстоянием для него же, начинаем основной цикл, на каждой итерации которого берем ближайший узел current\_node с расстоянием current\_dist, проверяем является ли узел целевым, если да то собираем ответ и выходим, если нет, то обходим каждого из current\_node->nodes пробуя улучшить расстояние, и если находим более короткий путь, то обновляем distances, came\_from и pq. Если цикл завершился вследствие того что очередь пуста, то значит что путь не существует, выходим с пустым путем и бесконечным расстоянием.

Для реализации поиска кратчайшего пути в графе для алгоритма А\* написан метод astar\_shortest. Также используется эвристика distance\_between оценивающая расстояние прямым расстоянием на координатной плоскости. После начальных проверок создаются переменные для обхода – open\_set – приоритетная очередь для пары суммарная стоимость-узел, для автоматического получения ближайшего узла. g\_cost – наилучшее известное расстояние от старта до узлов. f\_cost – предполагаемое расстояние от старта до узлов. came\_from – контейнер для накопления пути найденного в процессе обхода. После начальной инициализации g\_cost нулевым расстоянием, f\_cost оценочным расстоянием до целевого узла, open\_set начальным узлом и оценочным расстоянием, начинаем основной цикл. На каждой итерации которого производим следующее: извлекаем узел current с минимальным f\_cost, если искомый путь найдем – формируем ответ, если нет, то для каждого соседа проверяем нашли ли мы лучший путь путем проверки значений tentative\_g\_cost < g\_cost[neighbor] и если лучший путь найден, то обновляем значения g\_cost, f\_cost, came\_from, выставляем значение в open\_set. Если цикл завершился вследствие того что очередь пуста, то значит что путь не существует, выходим с пустым путем и бесконечным расстоянием.

1. Экспериментальная часть

Для работы алгоритма производится загрузка графа из предоставленного файла, затем поиск узлов ближайших к координатам 30.30777, 59.95497 для перекрестка дорог рядом с ИТМО, и 30.382467, 59.979395 для перекрестка дорого около моего дома. После нахождения узлов последовательно запускаются алгоритмы, выводящие в консоль результаты выполнения в виде количества найденных узлов, и длины найденного пути, для BFS и DFS расстояние считается в единицах ребер графа.

Результаты работы:



Изображение №1 – результаты и время выполнения в миллисекундах для алгоритма BFS



Изображение №2 – результаты и время выполнения в миллисекундах для алгоритма DFS



Изображение №3 – результаты и время выполнения в миллисекундах для алгоритма Дейкстры



Изображение №4 – результаты и время выполнения в миллисекундах для алгоритма А\*

Подсчёт по памяти (только для циклов и сложных структур).

bfs\_shortest – O(V)

dfs – O(V)

dijkstra\_shortest – O(V)

astar\_shortest – O(V)

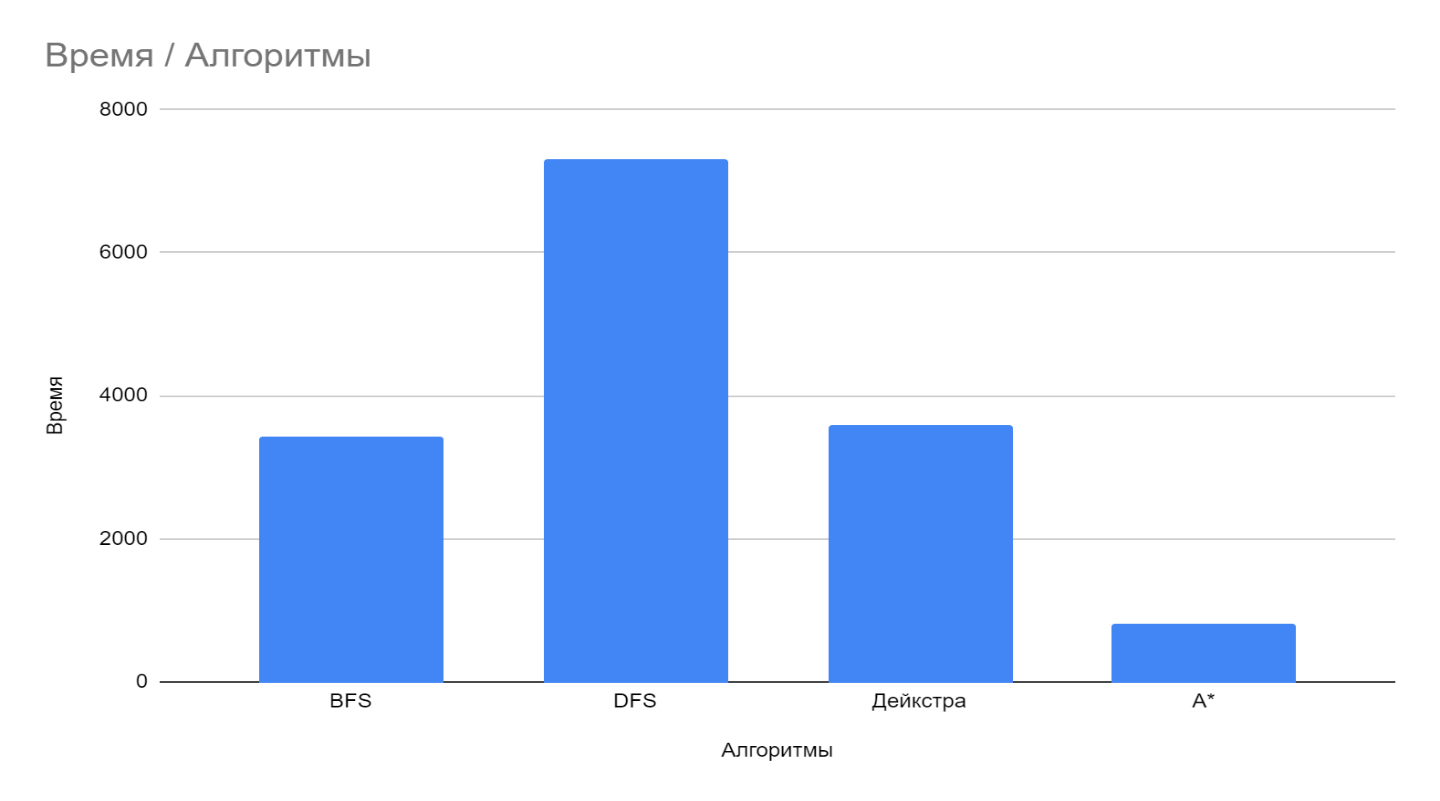
Подсчёт асимптотики (только для циклов и сложных структур)

bfs\_shortest – O(V + E)

dfs – O(V + E)

dijkstra\_shortest – O((V + E) \* log V)

astar\_shortest – O((V + E) \* log V)



Изображение №5 – График сравнения времени работы алгоритмов поиска пути в графе

1. Заключение

Алгоритм А\* для заданного графа и пары значений, показал наилучший результат, превосходящий и DFS и BFS но и алгоритм Дейкстры. DFS – слишком медленный и не оптимальный для задач на большом графе.

Рассмотренные алгоритмы могут быть использованы для разных целей, BFS – для поиска в случае если граф не имеет весов, А\* для поиска оптимальных путей в графах, особенно подходит для больших графов, благодаря своей хорошей асимптотической сложности, DFS – полезен для поиска любого пути, не обязательно оптимального, что может быть полезно для проверки связности, Дейкстра в целом сравним в А\* и подходит для сравнимых задач, но А\* может обогнать его за счет хорошей эвристики, что и произошло в данном случае.

Таким образом все рассмотренные алгоритмы являются полезными в своей области применения, несмотря на различия в скорости в конкретной рассмотренной задаче. Но в общем для поиска кратчайшего пути наилучшим алгоритмом будет выступать A\*, с асимптотической сложностью (V + E) \* log V, как и у алгоритма Дейкстры, но засчёт эвристики, A\* не реализует бесполезные проходы по минимальному весу, которые есть в Дейкстре, но и учитывает расстояние до нужного узла. Dfs и bfs эффективны для полного обхода графа, с малым количеством вершин и рёбер, но не для вывода кратчайшего пути в большом графе, за счёт резко возрастающей асимптотики. Если задачей является найти кратчайший путь в невзвешенном связном графе, особенно, если он небольшой, то bfs будет хорошим вариантом для поставленной задачи.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла graph.cpp

#include <iostream>

#include <vector>

#include <chrono>

#include <cmath>

#include <limits>

#include <fstream>

#include <string>

#include <sstream>

#include <unordered\_map>

#include <queue>

#include <unordered\_set>

#include <stack>

// Глобальные переменные для хранения временных меток

auto start = std::chrono::steady\_clock::now();

    // std::chrono::steady\_clock::time\_point обычно ~8 байт (зависит от реализации).

    // O(1) по времени на инициализацию (runtime), O(1) по памяти

auto end = std::chrono::steady\_clock::now();

    // То же самое (~8 байт).

    // O(1) по времени, O(1) по памяти

// Функция запуска счётчика времени

void start\_counter() {

    // Локальная операция присвоения: сама функция не выделяет память,

    // перезаписывает значение глобальной переменной (start).

    // ~8 байт уже храним в глобальном объекте.

    // O(1) по времени, O(1) по памяти

    start = std::chrono::steady\_clock::now();

}

// Функция остановки счётчика времени и вывода результата

void stop\_counter(std::string algorithm) {

    // Пара локальных переменных:

    // 1) end - глобальная переменная (перезаписываем ~8 байт) -> O(1) по памяти

    // 2) elapsed - 64-битное целое (std::chrono::milliseconds::rep), обычно 8 байт -> O(1) по памяти

    // O(1) по времени: получение времени, вывод в поток

    end = std::chrono::steady\_clock::now();

    auto elapsed = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

    // ~8 байт для счётчика

    // O(1) по времени, O(1) по памяти

    std::cout << algorithm <<" Execution time: " << elapsed << std::endl;

    // Вывод в поток, временные строки в стеке

    // O(1) по времени, O(1) по памяти

}

// Функция splitString: разбивает строку по delimiter и возвращает vector<string>.

std::vector<std::string> splitString(const std::string& str, char delimiter) {

    // result - динамический массив std::vector<std::string>, начальное выделение обычно под маленькую ёмкость

    // (например, 0 или 1-2 элементов), затем увеличение по мере добавления.

    // Для каждого добавленного элемента выделяется память под std::string (обычно ~24-32 байта «на местах» + динамическая память под сам текст).

    // O(1) по памяти на создание самого объекта result (он пустой), далее - динамический рост

    // В худшем случае, если в строке будет k разделителей, результирующий вектор size() = k+1

    std::vector<std::string> result;

    // O(1) по времени, O(1) по памяти

    std::stringstream ss(str); // std::stringstream внутри себя содержит буфер (динамическая память под копию str).

                               // O(L) по времени, где L = длина строки, чтобы скопировать внутрь (если реализовано так),

                               // O(L) по памяти под буфер

    // Локальная std::string (~24-32 байта под управляющие структуры + динамическая память при росте строки).

    // O(1) по времени (создание пустой строки), O(1) по памяти (управляющая структура)

    std::string item;

    // Цикл читает из ss и кладёт во временный item.

    // При push\_back(item) в result создаются копии std::string.

    // По времени: O(L) для прохода всей строки

    // (каждый вызов std::getline занимает O(m) для выделенного участка).

    // По памяти: каждая новая std::string может потребовать O(m) динамической памяти (где m = длина сегмента).

    while (std::getline(ss, item, delimiter)) {

        // O(m) по времени на каждую итерацию при чтении сегмента,

        // O(1) амортизированно на вставку в vector (пока не превышена ёмкость).

        result.push\_back(item);

        // O(m) по памяти (содержимое строки) плюс ~24-32 байта на служебную часть std::string

    }

    return result;

    // Возвращается по RVO/NRVO или move, доп. памяти не выделяется.

    // В целом асимптотика по времени: O(L) (где L = длина `str`),

    // по памяти: O(L) (размер под итоговые подстроки).

}

// Структура Node: хранит идентификатор, координаты и список соседей.

struct Node {

    int id;       // int обычно 4 байта.

                  // O(1) по памяти

    double lon;   // double обычно 8 байт.

                  // O(1) по памяти

    double lat;   // double обычно 8 байт.

                  // O(1) по памяти

    // Вектор пар (Node\*, double). Каждый std::pair<Node\*, double> обычно =

    // (8 байт под Node\*, 8 байт под double) = 16 байт на пару, плюс служебная память std::vector.

    // Динамически растёт по мере добавления соседей.

    // O(1) при создании пустого, O(k) по памяти при наличии k соседей.

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

};

// Функтор сравнения для очереди с приоритетами

struct CompareNode {

    // Оператор(), принимает две пары (double, Node\*) и сравнивает first.

    // Сам функтор не хранит полей, значит не занимает память на уровне экземпляра,

    // только в момент вызова идёт сравнение.

    // O(1) по времени на каждое сравнение, O(1) по памяти

    bool operator()(const std::pair<double, Node\*>& a, const std::pair<double, Node\*>& b) const {

        return a.first > b.first; // O(1) сравнение

    }

};

// Структура Graph: хранит узлы в хеш-таблице и реализует алгоритмы поиска путей.

struct Graph {

    // Основное поле — std::unordered\_map<std::string, Node\*>

    // - для каждой пары (ключ-значение) хранится ~ (хэш + key + value) в хеш-таблице.

    // - ключ (std::string) может занимать динамическую память (зависит от длины строки).

    // - value (Node\*) — 8 байт (указатель).

    // О(1) средняя вставка/поиск, О(N) в худшем случае (коллизии).

    // По памяти: O(N \* (size of string + 8 байт на Node\*)) + служебные поля

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

    int max\_id\_current = 0; // 4 байта (int).

                            // O(1) по памяти

    // Создаём или возвращаем существующий узел по строке координат

    Node\* getOrAddNodeByCoordinatesLine(std::string coordinates\_string) {

        // Локальная переменная it — это итератор std::unordered\_map, занимает не более ~8-16 байт на стеке.

        // Поиск в хеш-таблице: O(1) в среднем, O(N) в худшем

        auto it = this->nodes.find(coordinates\_string);

        if (it != this->nodes.end()) {

            return it->second;

            // Если нашли, не выделяем ничего нового

            // O(1) по времени, O(1) по памяти

        }

        // Создаём новый узел (Node).

        // Выделение памяти в куче под сам Node.

        // O(1) по времени (амортизированно), O(1) по памяти (фиксированный размер Node)

        Node\* current\_node = new Node();

        // splitString вернёт вектор строк;

        // в зависимости от длины входной строки будет несколько std::string в vector.

        // O(L) по времени, O(L) по памяти

        std::vector<std::string> current\_coordinates = splitString(coordinates\_string, ',');

        current\_node->id = this->max\_id\_current;

        // O(1) по времени, O(1) по памяти (id уже внутри Node)

        current\_node->lon = std::stod(current\_coordinates[0]);

        // std::stod может аллоцировать временные объекты в стеке, O(1) по времени для короткой строки

        current\_node->lat = std::stod(current\_coordinates[1]);

        current\_node->nodes = std::vector<std::pair<Node\*, double>>();

            // Пустой вектор (минимальное внутреннее хранилище, обычно 0..16 байт).

            // O(1) по времени, O(1) по памяти

        // Запись в хеш-таблицу: хеш-таблица может перераспределяться (re-hash),

        // выделяя память под новые бакеты.

        // В среднем O(1) по времени, O(N) в худшем.

        // По памяти: храним key (string) + value (Node\*) + служебный узел.

        this->nodes[coordinates\_string] = current\_node;

        this->max\_id\_current++;

        // инкремент (4 байта int).

        // O(1) по времени, O(1) по памяти

        return current\_node;

        // O(1) по времени, O(1) по памяти

    }

    // Находит среди всех узлов тот, который ближе всего к (lon, lat)

    Node\* findClosestNode(double lon, double lat,

                          double min\_distance = std::numeric\_limits<double>::max()) {

        // node\_founded — локальный указатель (8 байт на стеке).

        // O(1) по памяти

        Node\* node\_founded = nullptr;

        // Проходим по всей хеш-таблице (nodes).

        // O(N) по времени, где N — кол-во узлов

        // Нет новых крупных аллокаций, только чтение данных из map.

        for (const auto &current : nodes) {

            // O(1) извлечение second

            Node\* node = current.second; // 8-байтовый указатель

            double distance = std::sqrt(std::pow(node->lat - lat, 2) +

                                        std::pow(node->lon - lon, 2));

            // O(1) по времени вычисление

            // Локальная double переменная distance (~8 байт в стеке).

            if (distance < min\_distance) {

                node\_founded = node;

                min\_distance = distance;

                // O(1)

            }

        }

        return node\_founded;

        // O(1) по памяти

    }

    // Вспомогательный метод для вычисления евклидова расстояния

    double distance\_between(const Node\* a, const Node\* b) {

        // Возвращает double (~8 байт), вычисляет sqrt(...).

        // O(1) по времени, O(1) по памяти

        return std::sqrt(std::pow(a->lon - b->lon, 2) + std::pow(a->lat - b->lat, 2));

    }

    // Восстановление пути по map `came\_from`

    // Возвращает вектор int (id), динамически аллоцированный внутри std::vector.

    std::vector<int> reconstructPath(

        const std::unordered\_map<int, int>& came\_from,

        const int& start\_node\_id, const int& goal\_node\_id)

    {

        // path - вектор int; каждый int ~4 байта. Размер зависит от длины пути.

        // O(k) по памяти, где k — длина пути

        std::vector<int> path;

        // Цикл идёт от goal\_node\_id назад к start\_node\_id,

        // используя came\_from (хеш-таблица). В среднем доступ O(1), худший O(k).

        // В худшем случае k ~ N, если путь проходит через все узлы.

        for (int at = goal\_node\_id;

             at != start\_node\_id;

             at = came\_from.at(at))

        {

            path.push\_back(at); // O(1) амортизированно при вставке в вектор

        }

        path.push\_back(start\_node\_id);

        // O(1)

        // Реверс вручную

        std::vector<int> reversed\_path(path.size());

        // Выделение массива size() \* 4 байта под int. O(k) по памяти

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

            reversed\_path[i] = path[j];

            // O(1) в каждой итерации

        }

        return reversed\_path;

        // RVO/мув

        // Итог: O(k) по времени и памяти, где k — длина пути

    }

    // Возвращает g\_cost[node\_id], если он есть, иначе бесконечность

    double getGCost(const std::unordered\_map<int, double>& g\_cost,

                    const int& node\_id)

    {

        // В среднем O(1) доступ, в худшем O(V) (коллизии)

        // Возвращается double (8 байт).

        auto it = g\_cost.find(node\_id);

        if (it == g\_cost.end()) {

            return std::numeric\_limits<double>::infinity();

        }

        return it->second;

    }

    //---------------------------------------------------------//

    // 1) BFS — Возвращаем (путь, стоимость)

    //    Стоимость = кол-во рёбер (граф невзвешенный).

    // По времени - O(V + E)

    // По памяти - O(V

    //---------------------------------------------------------//

    std::pair<std::vector<int>, double> bfs\_shortest(Node\* start\_node, Node\* goal\_node)

    {

        // Стартуем счётчик времени

        // O(1)

        start\_counter();

        // Если старт и цель — один и тот же узел

        if (start\_node->id == goal\_node->id) {

            stop\_counter("BFS");

            // Возвращаем путь, состоящий из одного узла, и стоимость 0

            // O(1)

            return {{start\_node->id}, 0};

        }

        // Подготовка структур:

        // queue<Node\*>    — динамическая структура, хранит указатели на Node. 8 байт на указатель Node + 40-100 байт сама очередь

        // came\_from       — хеш-таблица int->int. 24-32 байта на каждый добавленный элемент, включая служебную информацию + 64 - 80 байт на пустую хэш таблицу

        // distances       — хеш-таблица int->double. 24-32 байта на каждый добавленный элемент, включая служебную информацию + 64 - 80 байт на пустую хэш таблицу

        // visited         — хеш-сет int. 16-24 байта на каждый добавленный элемент, включая служебную информацию + 64 - 80 байт на пустую хэш таблицу

        // По времени: Все операции с хеш-таблицами и очередью — в среднем O(1) на вставку/извлечение,

        // BFS в целом O(V + E).

        // По памяти: O(V) на visited, came\_from, distances; O(V) на queue в худшем случае.

        std::queue<Node\*> queue;

        std::unordered\_map<int, int> came\_from;

        std::unordered\_map<int, double> distances;

        std::unordered\_set<int> visited;

        // Присвоение в таблицу и множество

        // O(1) в среднем

        distances[start\_node->id] = 0.0;

        visited.insert(start\_node->id);

        queue.push(start\_node);

        // Основной цикл BFS

        while (!queue.empty()) {

            Node\* current\_node = queue.front(); // 8 байт

            // O(1)

            queue.pop();

            // O(1)

            if (current\_node->id == goal\_node->id) {

                double cost = distances[goal\_node->id]; // 8 байт

                // Достаём double из map, 8 байт

                auto path = reconstructPath(came\_from, start\_node->id, goal\_node->id);

                // path — std::vector<int>, 4 байта на int + 20-32 байт сам вектор

                stop\_counter("BFS");

                return { path, cost };

            }

            // Перебираем всех соседей current\_node

            // current\_node->nodes: std::vector<std::pair<Node\*, double>>

            // Каждый push может аллоцировать память в queue.

            // Суммарно по всем узлам: O(V + E) для обхода.

            for (const auto& neighbor : current\_node->nodes) {

                Node\* neighbor\_node = neighbor.first; // 8 байт на указатель Node

                if (visited.find(neighbor\_node->id) == visited.end()) {

                    // O(1) в среднем

                    visited.insert(neighbor\_node->id);

                    came\_from[neighbor\_node->id] = current\_node->id;

                    distances[neighbor\_node->id] = distances[current\_node->id] + 1.0;

                    queue.push(neighbor\_node);

                }

            }

        }

        stop\_counter("BFS");

        // Если путь не найден

        // O(1)

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

    }

    //---------------------------------------------------------//

    // 1) DFS — Возвращаем (путь, стоимость)

    //    Стоимость = кол-во рёбер (граф невзвешенный).

    // По времени - O(V + E)

    // По памяти - O(V)

    //---------------------------------------------------------//

    std::pair<std::vector<int>, double> dfs(Node\* start\_node, Node\* goal\_node)

    {

        start\_counter();

        // O(1)

        if (start\_node->id == goal\_node->id) {

            stop\_counter("DFS");

            // O(1)

            return {{start\_node->id}, 0};

        }

        // Стек для итеративного DFS

        // Хранит указатели на Node 8 байт на каждый элемент + 40-100 байт на сам контейнер (std::deque) внутри stack.

        // O(V) потенциально в худшем случае

        std::stack<const Node\*> st;

        // Набор (hash set) по id (int). 16-24 байт на элемент, включая служебную информацию + 64-80 на сам set

        // O(V) в худшем случае

        std::unordered\_set<int> visited;

        // Карта предков (int->int). 24-32 байт на элемент, включая служебную информацию + 64-80 на сам map

        // O(V) в худшем случае

        std::unordered\_map<int, int> came\_from;

        // Инициализация

        // O(1)

        st.push(start\_node);

        visited.insert(start\_node->id);

        // По времени: в худшем случае DFS O(V + E).

        while (!st.empty()) {

            const Node\* current = st.top(); // 8 байт

            st.pop();

            if (current == goal\_node) {

                // Восстанавливаем путь

                // O(k), где k — длина пути

                std::vector<int> path = reconstructPath(came\_from, start\_node->id, goal\_node->id);

                // path — std::vector<int>, 4 байта на int + 20-32 байт сам вектор

                double edge\_count = static\_cast<double>(path.size() - 1); // 8 байт

                stop\_counter("DFS");

                return { path, edge\_count };

            }

            // Смотрим всех соседей

            // O(deg(current)) за одну итерацию; суммарно O(E) для всего обхода

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

                const Node\* neighbor = edge.first;  // 8 байт на указатель Node

                if (!neighbor) continue;

                if (visited.find(neighbor->id) == visited.end()) {

                    visited.insert(neighbor->id);

                    came\_from[neighbor->id] = current->id;

                    st.push(neighbor);

                }

            }

        }

        stop\_counter("DFS");

        // Путь не найден

        // O(1)

        return {{}, 0.0};

    }

    //---------------------------------------------------------//

    // 3) Ленивая Dijkstra — (путь, суммарный вес)

    // По времени - O(E \* log V) в худшем случае

    // По памяти - O(V)

    //---------------------------------------------------------//

    std::pair<std::vector<int>, double> dijkstra\_shortest(Node\* start\_node, Node\* goal\_node)

    {

        start\_counter();

        // O(1)

        if (start\_node->id == goal\_node->id) {

            stop\_counter("Dijkstra");

            return {{start\_node->id}, 0};

        }

        // Приоритетная очередь (мин-куча) хранит пары (double, Node\*)

        // - внутри себя vector<std::pair<double, Node\*>>

        // - каждое помещение ~O(log V) в худшем случае

        // - хранение ~ (16 байт на элемент = 8 double + 8 указатель на Node) + 20-40 байт сама очередь

        std::priority\_queue<

            std::pair<double, Node\*>,

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

            CompareNode

        > pq;

        // distances — hash map: int -> double. 24-32 на элемент с учетом служебной информации + 64-80 на сам map

        // came\_from — hash map: int -> int. 24-32 на элемент с учетом служебной информации +  64-80 на сам map

        // По времени: Dijkstra — O(E log V) в худшем случае (при «ленивой» проверке).

        // По памяти: O(V) на хранение distance, came\_from, + O(V) в приоритетной очереди в худшем случае.

        std::unordered\_map<int, double> distances;

        std::unordered\_map<int, int> came\_from;

        distances[start\_node->id] = 0.0;

        pq.emplace(0.0, start\_node);

        // Dijkstra: O(E log V) в худшем случае

        while (!pq.empty()) {

            double current\_dist = pq.top().first; // 8 байт

            Node\* current\_node  = pq.top().second; // 8 байт на указатель Node

            pq.pop();

            double best\_known\_for\_current = getGCost(distances, current\_node->id); // 8 байт

            // O(1) среднее

            if (current\_dist > best\_known\_for\_current) {

                continue;

                // Ленивая проверка

                // O(1)

            }

            if (current\_node == goal\_node) {

                std::vector<int> path = reconstructPath(came\_from, start\_node->id, goal\_node->id);

                // path — std::vector<int>, 4 байта на int + 20-32 байт сам вектор, O(k)

                double cost = getGCost(distances, goal\_node->id); // 8 байт

                // O(1)

                stop\_counter("Dijkstra");

                return { path, cost };

            }

            // Перебираем всех соседей

            // Суммарно за всё время: O(E) итераций

            for (auto& edge : current\_node->nodes) {

                Node\* neighbor\_node = edge.first; // 8 байт

                double edge\_weight  = edge.second; // 8 байт

                double old\_dist = getGCost(distances, neighbor\_node->id); // 8 байт

                // O(1) среднее

                double new\_dist = current\_dist + edge\_weight; // 8 байт

                if (new\_dist < old\_dist) {

                    distances[neighbor\_node->id] = new\_dist;

                    came\_from[neighbor\_node->id] = current\_node->id;

                    pq.emplace(new\_dist, neighbor\_node);

                    // Вставка в очередь ~O(log V)

                }

            }

        }

        stop\_counter("Dijkstra");

        // Путь не найден

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

    }

    //---------------------------------------------------------//

    // 4) A\* (A-Star) — (путь, суммарный вес)

    // По времени - O(E\*log V) в худшем случае

    // По памяти - O(V)

    //---------------------------------------------------------//

    std::pair<std::vector<int>, double> astar\_shortest(Node\* start\_node, Node\* goal\_node)

    {

        start\_counter();

        // O(1)

        if (start\_node->id == goal\_node->id) {

            stop\_counter("A\*");

            return {{start\_node->id}, 0};

        }

        // open\_set — приоритетная очередь (fCost = gCost + heuristic).

        // O(log V) на вставку/извлечение

        std::priority\_queue<

            std::pair<double, Node\*>,

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

            CompareNode

        > open\_set;

        // g\_cost, f\_cost — hash map: int -> double. 24-32 на элемент с учетом служебной информации + 64-80 на сам map

        // came\_from — hash map: int -> int. 24-32 на элемент с учетом служебной информации +  64-80 на сам map

        // A\* ~ O(E log V) в худшем случае

        std::unordered\_map<int, double> g\_cost;

        std::unordered\_map<int, double> f\_cost;

        std::unordered\_map<int, int> came\_from;

        g\_cost[start\_node->id] = 0.0;

        f\_cost[start\_node->id] = distance\_between(start\_node, goal\_node);

        open\_set.emplace(f\_cost[start\_node->id], start\_node);

        // A\* ~ O(E log V) в худшем случае, зависит от эвристики

        while (!open\_set.empty()) {

            Node\* current = open\_set.top().second; // 8 байт

            open\_set.pop();

            if (current == goal\_node) {

                auto path = reconstructPath(came\_from, start\_node->id, goal\_node->id);

                // path — std::vector<int>, 4 байта на int + 20-32 байт сам вектор, O(k)

                double cost = g\_cost[goal\_node->id]; // 8 байт O(1)

                stop\_counter("A\*");

                return { path, cost };

            }

            // Перебираем все ребра из current

            // Суммарно за все итерации O(E)

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

                Node\* neighbor\_node = edge.first; // 8 байт

                double edge\_weight = edge.second; // 8 байт

                double current\_g = getGCost(g\_cost, current->id); // 8 байт

                double tentative\_g\_cost = current\_g + edge\_weight; // 8 байт

                double neighbor\_g = getGCost(g\_cost, neighbor\_node->id); // 8 байт

                if (tentative\_g\_cost < neighbor\_g) {

                    g\_cost[neighbor\_node->id] = tentative\_g\_cost;

                    double heuristic = distance\_between(neighbor\_node, goal\_node); // 8 байт

                    f\_cost[neighbor\_node->id] = tentative\_g\_cost + heuristic;

                    came\_from[neighbor\_node->id] = current->id;

                    open\_set.emplace(f\_cost[neighbor\_node->id], neighbor\_node);

                    // ~O(log V)

                }

            }

        }

        stop\_counter("A\*");

        // Если путь не найден

        // O(1)

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

    }

};

ПРИЛОЖЕНИЕ B

Листинг кода файла main.cpp

#include <iostream>

#include <vector>

#include <chrono>

#include <cmath>

#include <limits>

#include <fstream>

#include <string>

#include <sstream>

#include <unordered\_map>

#include <queue>

#include <unordered\_set>

#include <stack>

#include "graph.cpp"

Graph\* load\_graph\_from\_file(std::string file\_name) {

    std::string line;

    std::ifstream in(file\_name);

    Graph\* graph = new Graph();

    while(std::getline(in, line))

    {

        std::vector<std::string> graph\_node\_parts = splitString(line, ':');

        Node\* current\_node = graph->getOrAddNodeByCoordinatesLine(graph\_node\_parts[0]);

        std::vector<std::string> neighbours\_strings = splitString(graph\_node\_parts[1], ';');

        for (std::string neighbour\_string : neighbours\_strings) {

            auto neighbour\_parts = splitString(neighbour\_string, ',');

            auto neighbour\_key = neighbour\_parts[0] + "," + neighbour\_parts[1];

            auto neighbour\_weight = stod(neighbour\_parts[2]);

            Node\* child = graph->getOrAddNodeByCoordinatesLine(neighbour\_key);

            current\_node->nodes.emplace\_back(child, neighbour\_weight);

        }

    }

    return graph;

}

void test\_assert(std::pair<std::vector<int>, double> result, std::string test\_name, int expected\_length, double expected\_distance) {

    std::cout << test\_name << " Length:" << result.first.size() << " Distance:" << result.second << std::endl;

    if (result.first.size() == expected\_length && result.second == expected\_distance) {

        std::cout << test\_name << " PASSED" << std::endl;

    } else {

        std::cout << test\_name << " FAILED" << std::endl;

    }

}

void tests(Graph\* graph) {

    std::pair<std::vector<int>, double> result;

    auto itmo\_node = graph->findClosestNode(30.30777, 59.95497, 0.001);

    auto home\_node = graph->findClosestNode(30.382467, 59.979395, 0.001);//(30.374287, 59.980944, 0.001);//graph->findClosestNode(30.309108, 59.967238, 0.001);//graph->findClosestNode(30.374287, 59.980944, 0.001);

    //std::cout << "Src:" << home\_node->id << " Dst:" << itmo\_node->id << std::endl;

    result = graph->bfs\_shortest(home\_node, itmo\_node);

    test\_assert(result, "BFS", 118, 117);

    result = graph->bfs\_shortest(home\_node, home\_node);

    test\_assert(result, "BFS same node", 1, 0);

    result = graph->astar\_shortest(home\_node, itmo\_node);

    test\_assert(result, "A\*", 140, 0.10066647269198108);

    result = graph->astar\_shortest(home\_node, home\_node);

    test\_assert(result, "A\* same node", 1, 0);

    result = graph->dijkstra\_shortest(home\_node, itmo\_node);

    test\_assert(result, "Dijkstra", 140, 0.10066647269198108);

    result = graph->dijkstra\_shortest(home\_node, home\_node);

    test\_assert(result, "Dijkstra same node", 1, 0);

    result = graph->dfs(home\_node, itmo\_node);

    test\_assert(result, "DFS", 8983, 8982);

    result = graph->dfs(home\_node, itmo\_node);

    test\_assert(result, "DFS same node", 8983, 8982);

}

int main() {

    auto graph = load\_graph\_from\_file("spb\_graph.txt");

    tests(graph);

    return 0;

}