ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Алгоритмы поиска кратчайшего пути»

Выполнила работу

Шакина Анна

Академическая группа № J3110

Принято

Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

Цель: найти стандартный (кратчайший) путь от места проживания к одному из корпусов ИТМО.

Задачи:

1. Написать алгоритмы поиска пути (кратчайшего пути) между 2-мя точками в графе.
2. Зафиксировать время работы, сравнить его.
3. Теоретическая подготовка
   1. BFS.

Алгоритм 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.

* 1. DFS

Алгоритм 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.

* 1. Dijkstra

Используется для нахождения кратчайшего пути от одного узла до всех остальных узлов в взвешенном графе с положительными весами рёбер. Он использует приоритетную очередь для хранения узлов, которые нужно исследовать.

* 1. 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. Реализация
   1. Используемые библиотеки

* <vector>: используется для работы с динамическими массивами (векторами), которые могут изменять свой размер во время выполнения программы.
* <cmath>: предоставляет математические функции, такие как тригонометрические функции, функции возведения в степень и другие математические операции.
* <queue>: используется для работы с очередями, что позволяет реализовывать структуры данных, такие как очередь (FIFO) и приоритетная очередь.
* <unordered\_map>: предоставляет реализацию хеш-таблицы для хранения пар "ключ-значение", обеспечивая быстрый доступ к элементам по ключу.
* <unordered\_set>: предоставляет реализацию неупорядоченного множества, позволяя хранить уникальные элементы и обеспечивая быстрый доступ к ним.
* <limits>: используется для определения предельных значений типов данных, таких как максимальные и минимальные значения целых чисел и чисел с плавающей запятой.
* <fstream>: предоставляет функциональность для работы с файлами, включая чтение из файлов и запись в файлы.
* <sstream>: используется для работы со строковыми потоками, позволяя преобразовывать строки в другие типы данных и наоборот.
* <iostream>: предоставляет стандартные функции ввода-вывода, такие как std::cin и std::cout, для работы с консолью.
* <functional>: предоставляет функциональные объекты и функциональные возможности, такие как указатели на функции и лямбда-функции.
* <memory>: предоставляет умные указатели (например, std::unique\_ptr и std::shared\_ptr) для управления динамически выделенной памятью и предотвращения утечек памяти
* <stack>: используется для работы со стеком, что позволяет реализовать структуру данных LIFO (последний пришёл — первый вышел).
* <chrono>: предоставляет функциональность для измерения времени, что позволяет отслеживать продолжительность выполнения операций.
  1. Решение задачи

Для дальнейшей работы с графом необходимо создать структуры Node и Graph, а также хэш-функцию для координат. Подробное описание представлено в комментариях кода:

*// структура для представления узла графа*

struct **Node**

{

    double lon, lat; *// долгота и широта узла*

    std::**vector**<std::**pair**<**Node** \*, double>> edges; *// список рёбер, исходящих из узла*

};

*// хэш-функция для пар (долгота, широта) для использования в unordered\_map*

struct **pair\_hash**

{

*// шаблонный оператор для вычисления хэша пары значений*

    template <class **T1**, class **T2**>

**size\_t** **operator()**(const std::**pair**<**T1**, **T2**> &p) const

    {

        auto h1 = std::**hash**<**T1**>{}(p.first); *// вычисляем хэш первого элемента пары*

        auto h2 = std::**hash**<**T2**>{}(p.second); *// вычисляем хэш второго элемента пары*

        return h1 ^ (h2 << 1); *// возвращаем комбинированный хэш с использованием побитового XOR и сдвига*

    }

};

*// структура для представления графа*

struct **Graph**

{

    std::**unordered\_map**<std::**pair**<double, double>, std::**unique\_ptr**<**Node**>, **pair\_hash**> nodes; *// хранилище узлов графа*

*// метод для поиска ближайшего узла к заданным координатам (широта, долгота)*

**Node** \***find\_closest\_node**(double lat, double lon)

    {

        double min\_distance = std::**numeric\_limits**<double>::**max**(); *// инициализация минимального расстояния*

**Node** \*node\_founded = nullptr; *// указатель на найденный узел*

*// проход по всем узлам графа*

        for (const auto &node : nodes)

        {

*// вычисление евклидова расстояния до текущего узла*

            double distance = **sqrt**(**pow**(node.second**->**lat - lat, 2) + **pow**(node.second**->**lon - lon, 2));

            if (distance < min\_distance)

            { *// если найденное расстояние меньше минимального*

                node\_founded = node.second.**get**(); *// обновление указателя на найденный узел*

                min\_distance = distance; *// обновление минимального расстояния*

            }

        }

        return node\_founded; *// возвращение указателя на ближайший узел*

    }

*// метод для добавления нового узла в граф*

    void **add\_node**(double lon, double lat)

    {

*// проверка, существует ли узел с такими координатами*

        if (nodes.**find**({lat, lon}) **==** nodes.**end**())

        {

            auto node = std::**make\_unique**<**Node**>(); *// создание уникального указателя на новый узел*

            node**->**lon = lon; *// установка долготы узла*

            node**->**lat = lat; *// установка широты узла*

            nodes**[**{lat, lon}**]** **=** **move**(node); *// добавление узла в хранилище графа*

        }

    }

*// метод для добавления ребра между двумя узлами*

    void **add\_edge**(double lon1, double lat1, double lon2, double lat2, double distance)

    {

        auto it1 = nodes.**find**({lat1, lon1}); *// поиск первого узла по координатам*

        auto it2 = nodes.**find**({lat2, lon2}); *// поиск второго узла по координатам*

        if (it1 **!=** nodes.**end**() && it2 **!=** nodes.**end**())

        { *// если оба узла найдены*

**Node** \*node1 = it1**->**second.**get**(); *// получение указателя на первый узел*

**Node** \*node2 = it2**->**second.**get**(); *// получение указателя на второй узел*

            node1->edges.**emplace\_back**(node2, distance); *// добавление второго узла в список рёбер первого узла*

            node2->edges.**emplace\_back**(node1, distance); *// добавление первого узла в список рёбер второго узла (двустороннее ребро)*

        }

        else

        {

            std::cerr **<<** "Ошибка: ";

            if (it1 **==** nodes.**end**())

            {

                std::cerr **<<** "Узел 1 (" **<<** lon1 **<<** ", " **<<** lat1 **<<** ") не найден. ";

            }

            if (it2 **==** nodes.**end**())

            {

                std::cerr **<<** "Узел 2 (" **<<** lon2 **<<** ", " **<<** lat2 **<<** ") не найден.";

            }

            std::cerr **<<** std::**endl**;

        }

    }

*// метод для загрузки графа из файла*

    void **load\_graph\_from\_file**(const std::**string** &filename)

    {

        std::**ifstream** **file**(filename); *// открытие файла для чтения*

        if (!file.**is\_open**()) *// проверка, открылся ли файл*

        {

            std::cerr **<<** "Невозможно открыть файл: " **<<** filename **<<** std::**endl**;

            return;

        }

        std::**string** line; *// переменная для хранения текущей строки*

        while (std::**getline**(file, line))

        { *// чтение файла построчно*

            std::**stringstream** **ss**(line); *// создание потока из строки*

            std::**string** node\_part, neighbor\_part; *// переменные для хранения части строки с узлом и соседями*

**getline**(ss, node\_part, ':'); *// получение части строки до двоеточия (узел)*

            double lon1, lat1;

            if (**sscanf**(node\_part.**c\_str**(), "%lf,%lf", &lon1, &lat1) != 2)

            {

                std::cerr **<<** "Не удалось прочитть координаты узла: " **<<** node\_part **<<** std::**endl**;

                continue;

            }

**add\_node**(lon1, lat1); *// добавление узла в граф*

            while (**getline**(ss, neighbor\_part, ';'))

            { *// чтение соседей, разделенных точкой с запятой*

                double lon2, lat2, distance;

                if (**sscanf**(neighbor\_part.**c\_str**(), "%lf,%lf,%lf", &lon2, &lat2, &distance) != 3)

                {

                    std::cerr **<<** "Не удалось прочитать координаты соседа и расстояние: " **<<** neighbor\_part **<<** std::**endl**;

                    continue;

                }

**add\_node**(lon2, lat2); *// добавление соседа в граф*

**add\_edge**(lon1, lat1, lon2, lat2, distance); *// добавление ребра между узлом и соседом*

            }

        }

    }

Листинг кода 1 - Хранение графа и чтение данных из файла

* 1. BFS

Функция bfs реализует поиск в ширину (BFS) для нахождения пути от узла start к узлу finish. Она использует очередь для хранения узлов, множество для отслеживания посещенных узлов и словарь для восстановления пути. Начальный узел добавляется в очередь и помечается как посещенный. В цикле извлекается текущий узел, и если он является конечным, восстанавливается путь через родителей. Если текущий узел не конечный, обрабатываются его соседи, которые еще не были посещены: они добавляются в очередь и отмечаются как посещенные. Если конечный узел не найден, возвращается пустой путь.

Более подробную информацию и соответствующий код можно найти в листинге 2.

    std::**vector**<**Node** \*> **bfs**(**Node** \*start, **Node** \*finish)

    {

        std::**queue**<**Node** \*> q; *// очередь для хранения узлов для обхода*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов*

        std::**unordered\_set**<**Node** \*> visited; *// множество посещенных узлов*

        std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

        q.**push**(start); *// добавление стартового узла в очередь*

        visited.**insert**(start); *// пометка стартового узла как посещенного*

        parent**[**start**]** = nullptr; *// установка родителя стартового узла как nullptr*

        while (!q.**empty**())

        { *// пока очередь не пуста*

**Node** \*current = q.**front**(); *// получение текущего узла из очереди*

            q.**pop**(); *// удаление текущего узла из очереди*

            if (current == finish)

            { *// если достигнут конечный узел*

                while (current)

                {

                    path.**push\_back**(current); *// добавление текущего узла в путь*

                    current = parent**[**current**]**; *// переход к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// обратный порядок пути (от конечного к стартовому)*

                return path; *// возвращение найденного пути*

            }

            for (auto [neighbor, \_] : current->edges)

            { *// проход по всем соседям текущего узла*

                if (visited.**find**(neighbor) **==** visited.**end**())

                { *// если сосед еще не посещен*

                    visited.**insert**(neighbor); *// пометка соседа как посещенного*

                    parent**[**neighbor**]** = current; *// установка текущего узла как родителя соседа*

                    q.**push**(neighbor); *// добавление соседа в очередь для дальнейшего обхода*

                }

            }

        }

        return path; *// возвращение пустого пути, если конечный узел не найден*

    }

Листинг 2 - Алгоритм BFS

* 1. DFS

Функция dfs реализует поиск в глубину (DFS) для нахождения пути от узла start к узлу finish. Она использует стек для хранения узлов, множество для отслеживания посещенных узлов и словарь для восстановления пути. Начальный узел добавляется в стек и помечается как посещенный. В цикле извлекается верхний узел, и если он является конечным, восстанавливается путь через родителей. Если текущий узел не конечный, обрабатываются его соседи, которые еще не были посещены: они добавляются в стек и отмечаются как посещенные. Если конечный узел не найден, возвращается пустой путь.

Для более подробного понимания реализации алгоритма DFS с комментариями можно обратиться к соответствующему коду в листинге 3.

    std::**vector**<**Node** \*> **dfs**(**Node** \*start, **Node** \*finish)

    {

        std::**stack**<**Node** \*> s; *// создаем стек для хранения узлов во время обхода графа*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов (для восстановления пути)*

        std::**unordered\_set**<**Node** \*> visited; *// множество для хранения посещенных узлов*

        std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

        s.**push**(start); *// добавляем стартовый узел в стек для начала обхода*

        visited.**insert**(start); *// помечаем стартовый узел как посещенный*

        parent**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr (нет родителя)*

        while (!s.**empty**())

        { *// пока стек не пустой*

**Node** \*current = s.**top**(); *// получаем текущий узел из вершины стека*

            s.**pop**(); *// удаляем текущий узел из стека*

            if (current == finish)

            { *// если достигнут конечный узел*

                while (current)

                { *// восстанавливаем путь от конечного до стартового узла*

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = parent**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// оборачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, \_] : current->edges)

            { *// проходим по всем соседям текущего узла*

                if (visited.**find**(neighbor) **==** visited.**end**())

                { *// если сосед еще не посещен*

                    visited.**insert**(neighbor); *// помечаем соседа как посещенного*

                    parent**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                    s.**push**(neighbor); *// добавляем соседа в стек для дальнейшего обхода*

                }

            }

        }

        return path; *// возвращаем пустой путь, если конечный узел не найден*

    }

Листинг 3 - Алгоритм DFS

* 1. Алгоритм Дейкстры

Функция dijkstra реализует алгоритм Дейкстры для поиска кратчайшего пути в графе от узла start до узла finish. Она инициализирует приоритетную очередь и массивы для хранения расстояний и родительских узлов. Начальное расстояние до стартового узла устанавливается в 0, и он добавляется в очередь. В основном цикле извлекается узел с наименьшим расстоянием, проверяется, не превышает ли текущее расстояние известное, и, если узел конечный, восстанавливается путь. Для каждого соседа узла вычисляется новое расстояние, и если оно меньше известного, обновляются значения и сосед добавляется в очередь. В случае отсутствия пути возвращается пустой вектор.

С кодом алгоритма Дейкстры и комментариями можно ознакомиться в листинге 4.

  std::**vector**<**Node** \*> **dijkstra**(**Node** \*start, **Node** \*finish)

    {

        std::**priority\_queue**<std::**pair**<double, **Node** \*>, std::**vector**<std::**pair**<double, **Node** \*>>, std::**greater**<>> pq; *// создаем приоритетную очередь для хранения узлов с их расстояниями*

        std::**unordered\_map**<**Node** \*, double> distances; *// хранилище для расстояний от стартового узла до каждого узла*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов*

        for (auto &node : nodes)

        {

            distances**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем расстояния до всех узлов как бесконечность*

        }

        distances**[**start**]** = 0; *// расстояние до стартового узла равно 0*

        pq.**push**({0, start}); *// добавляем стартовый узел в очередь с расстоянием 0*

        parent**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr*

        while (!pq.**empty**())

        {

            auto [current\_dist, current] = pq.**top**(); *// получаем узел с наименьшим расстоянием из очереди*

            pq.**pop**(); *// удаляем этот узел из очереди*

            if (current\_dist > distances**[**current**]**)

                continue; *// если текущее расстояние больше уже известного, пропускаем этот узел*

            if (current == finish)

            {

                std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

                while (current)

                {

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = parent**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// оборачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, edge\_dist] : current->edges)

            {

                if (edge\_dist < 0)

                {

                    std::cerr **<<** "Ошибка: отрицательное ребро" **<<** std::**endl**;

                    continue; *// пропускаем отрицательное ребро*

                }

                double new\_dist = current\_dist + edge\_dist; *// вычисляем новое расстояние до соседа*

                if (new\_dist < distances**[**neighbor**]**)

                {

                    distances**[**neighbor**]** = new\_dist; *// обновляем расстояние до соседа*

                    pq.**push**({new\_dist, neighbor}); *// добавляем соседа в очередь с обновленным расстоянием*

                    parent**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                }

            }

        }

        return {}; *// возвращаем пустой вектор, если путь не найден*

    }

Листинг 4 - Алгоритм Дейкстры

* 1. A\*

Алгоритм A\* ищет кратчайший путь в графе, комбинируя фактическую стоимость пути и эвристическую оценку оставшегося расстояния. Он использует приоритетную очередь для обработки узлов с наименьшей оценкой f(n), где f(n) = g(n) + h(n). Узлы обрабатываются, пока не будет найден конечный узел или не исчерпаются все варианты. Если путь найден, он восстанавливается через родительские узлы; если нет — возвращается пустой результат.

Подробнее ознакомиться с алгоритмом можно в листинге 5.

*// вычисление расстояния между двумя узлами*

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

    {

        return **sqrt**(**pow**(a->lat - b->lat, 2) + **pow**(a->lon - b->lon, 2)); *// вычисляем эвристическую функцию*

    }

    std::**vector**<**Node** \*> **a\_star**(**Node** \*start, **Node** \*finish)

    {

        std::**priority\_queue**<std::**pair**<double, **Node** \*>, std::**vector**<std::**pair**<double, **Node** \*>>, std::**greater**<>> open\_set; *// приоритетная очередь для хранения открытых узлов с их оценками f(n)*

        std::**unordered\_map**<**Node** \*, double> g\_score; *// хранилище для g(n) - стоимости пути от стартового узла до каждого узла*

        std::**unordered\_map**<**Node** \*, double> f\_score; *// хранилище для f(n) - полной стоимости пути через узел (g(n) + h(n))*

        std::**unordered\_map**<**Node** \*, **Node** \*> came\_from; *// хранилище для отслеживания родительских узлов*

*// инициализируем g(n) и f(n) для всех узлов как бесконечность*

        for (const auto &node : nodes)

        {

            g\_score**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем g(n) как бесконечность*

            f\_score**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем f(n) как бесконечность*

        }

        g\_score**[**start**]** = 0; *// стоимость пути от стартового узла до самого себя равна 0*

        f\_score**[**start**]** = **heuristic**(start, finish); *// вычисляем начальное значение f(n) для стартового узла*

        open\_set.**push**({f\_score**[**start**]**, start}); *// добавляем стартовый узел в открытый список с его оценкой f(n)*

        came\_from**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr*

        while (!open\_set.**empty**())

        {

**Node** \*current = open\_set.**top**().second; *// получаем узел с наименьшей оценкой f(n)*

            open\_set.**pop**(); *// удаляем этот узел из открытого списка*

            if (current == finish)

            {

                std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

                while (current)

                {

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = came\_from**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// разворачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, weight] : current->edges) *// проходим по всем соседям текущего узла*

            {

                if (weight < 0)

                {

                    std::cerr **<<** "Ошибка: отрицательное ребро" **<<** std::**endl**;

                    continue; *// пропускаем отрицательное ребро*

                }

                double tentative\_g\_score = g\_score**[**current**]** + weight; *// вычисляем предполагаемую стоимость пути до соседа через текущий узел*

                if (tentative\_g\_score < g\_score**[**neighbor**]**)

                {

                    came\_from**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                    g\_score**[**neighbor**]** = tentative\_g\_score; *// обновляем g(n) для соседа*

                    f\_score**[**neighbor**]** = g\_score**[**neighbor**]** + **heuristic**(neighbor, finish); *// обновляем f(n) для соседа*

                    open\_set.**push**({f\_score**[**neighbor**]**, neighbor}); *// добавляем соседа в открытый список с обновленным значением f(n)*

                }

            }

        }

        return {}; *// возвращаем пустой вектор, если путь не найден*

    }

Листинг 5 - Алгоритм A\*

1. Экспериментальная часть

После реализации алгоритмов было зафиксировано время их выполнения, а так же длина пути, с которыми можно ознакомиться на рисунке 1.

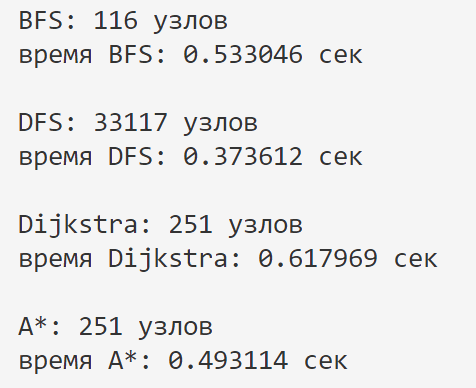


Рисунок 1 - Результат работы алгоритмов

BFS:

Алгоритм не учитывает веса рёбер при поиске пути и ориентирован на количество рёбер в невзвешенных графах, что позволяет находить оптимальный путь по количеству рёбер, делая обход в ширину. Однако его время работы больше других из-за необходимости исследовать все соседние узлы на каждом уровне.

DFS:

Алгоритм тоже не учитывает веса рёбер и исследует граф, углубляясь в него до тех пор, пока не достигнет конца пути или не найдет целевую вершину. Он не находит оптимальный путь, а просто обходит граф в глубину. Время работы наименьшее среди всех алгоритмов, так как DFS может завершить поиск быстрее, чем BFS, если целевая вершина находится глубоко в графе.

Дейкстра:

Алгоритм учитывает веса рёбер и находит кратчайший путь от начальной вершины ко всем остальным вершинам в графе с положительными весами. Время работы больше, чем у BFS, что говорит о том, что алгоритм нашел оптимальный путь, используя информацию о весах рёбер, а не только их количество.

A\*:

Алгоритм A\* также учитывает веса рёбер и применяет эвристику для ускорения поиска кратчайшего пути. Время работы быстрее Дейкстры, что может означать, что эвристика была правильно подобрана для данного графа, что позволило эффективно находить оптимальные пути. Этот алгоритм является лучшим для поиска минимального пути между вершинами в графе.

1. Заключение

Были разработаны и реализованы алгоритмы поиска кратчайшего пути в графе, включая BFS, DFS, Дейкстра и A\*. Каждый из них имеет свои особенности и области применения. Было проведено тестирование этих алгоритмов, результаты которого показали различное время выполнения и длину пути для разных алгоритмов. Наилучшую производительность показал алгоритм A\*, который благодаря использованию эвристики смог быстро найти оптимальный путь.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab8.cpp

#include <vector>

#include <cmath>

#include <queue>

#include <unordered\_map>

#include <unordered\_set>

#include <limits>

#include <fstream>

#include <sstream>

#include <iostream>

#include <functional>

#include <memory>

#include <stack>

#include <chrono>

*// структура для представления узла графа*

struct **Node**

{

    double lon, lat; *// долгота и широта узла*

    std::**vector**<std::**pair**<**Node** \*, double>> edges; *// список рёбер, исходящих из узла*

};

*// хэш-функция для пар (долгота, широта) для использования в unordered\_map*

struct **pair\_hash**

{

*// шаблонный оператор для вычисления хэша пары значений*

    template <class **T1**, class **T2**>

**size\_t** **operator()**(const std::**pair**<**T1**, **T2**> &p) const

    {

        auto h1 = std::**hash**<**T1**>{}(p.first); *// вычисляем хэш первого элемента пары*

        auto h2 = std::**hash**<**T2**>{}(p.second); *// вычисляем хэш второго элемента пары*

        return h1 ^ (h2 << 1); *// возвращаем комбинированный хэш с использованием побитового XOR и сдвига*

    }

};

*// структура для представления графа*

struct **Graph**

{

    std::**unordered\_map**<std::**pair**<double, double>, std::**unique\_ptr**<**Node**>, **pair\_hash**> nodes; *// хранилище узлов графа*

*// метод для поиска ближайшего узла к заданным координатам (широта, долгота)*

**Node** \***find\_closest\_node**(double lat, double lon)

    {

        double min\_distance = std::**numeric\_limits**<double>::**max**(); *// инициализация минимального расстояния*

**Node** \*node\_founded = nullptr; *// указатель на найденный узел*

*// проход по всем узлам графа*

        for (const auto &node : nodes)

        {

*// вычисление евклидова расстояния до текущего узла*

            double distance = **sqrt**(**pow**(node.second**->**lat - lat, 2) + **pow**(node.second**->**lon - lon, 2));

            if (distance < min\_distance)

            { *// если найденное расстояние меньше минимального*

                node\_founded = node.second.**get**(); *// обновление указателя на найденный узел*

                min\_distance = distance; *// обновление минимального расстояния*

            }

        }

        return node\_founded; *// возвращение указателя на ближайший узел*

    }

*// метод для добавления нового узла в граф*

    void **add\_node**(double lon, double lat)

    {

*// проверка, существует ли узел с такими координатами*

        if (nodes.**find**({lat, lon}) **==** nodes.**end**())

        {

            auto node = std::**make\_unique**<**Node**>(); *// создание уникального указателя на новый узел*

            node**->**lon = lon; *// установка долготы узла*

            node**->**lat = lat; *// установка широты узла*

            nodes**[**{lat, lon}**]** **=** **move**(node); *// добавление узла в хранилище графа*

        }

    }

*// метод для добавления ребра между двумя узлами*

    void **add\_edge**(double lon1, double lat1, double lon2, double lat2, double distance)

    {

        auto it1 = nodes.**find**({lat1, lon1}); *// поиск первого узла по координатам*

        auto it2 = nodes.**find**({lat2, lon2}); *// поиск второго узла по координатам*

        if (it1 **!=** nodes.**end**() && it2 **!=** nodes.**end**())

        { *// если оба узла найдены*

**Node** \*node1 = it1**->**second.**get**(); *// получение указателя на первый узел*

**Node** \*node2 = it2**->**second.**get**(); *// получение указателя на второй узел*

            node1->edges.**emplace\_back**(node2, distance); *// добавление второго узла в список рёбер первого узла*

            node2->edges.**emplace\_back**(node1, distance); *// добавление первого узла в список рёбер второго узла (двустороннее ребро)*

        }

    }

*// метод для загрузки графа из файла*

    void **load\_graph\_from\_file**(const std::**string** &filename)

    {

        std::**ifstream** **file**(filename); *// открытие файла для чтения*

        std::**string** line; *// переменная для хранения текущей строки*

        while (std::**getline**(file, line))

        { *// чтение файла построчно*

            std::**stringstream** **ss**(line); *// создание потока из строки*

            std::**string** node\_part, neighbor\_part; *// переменные для хранения части строки с узлом и соседями*

**getline**(ss, node\_part, ':'); *// получение части строки до двоеточия (узел)*

            double lon1, lat1;

**sscanf**(node\_part.**c\_str**(), "%lf,%lf", &lon1, &lat1); *// парсинг координат узла из строки*

**add\_node**(lon1, lat1); *// добавление узла в граф*

            while (**getline**(ss, neighbor\_part, ';'))

            { *// чтение соседей, разделенных точкой с запятой*

                double lon2, lat2, distance;

**sscanf**(neighbor\_part.**c\_str**(), "%lf,%lf,%lf", &lon2, &lat2, &distance); *// парсинг координат соседа и расстояния из строки*

**add\_node**(lon2, lat2); *// добавление соседа в граф*

**add\_edge**(lon1, lat1, lon2, lat2, distance); *// добавление ребра между узлом и соседом*

            }

        }

    }

    std::**vector**<**Node** \*> **bfs**(**Node** \*start, **Node** \*finish)

    {

        std::**queue**<**Node** \*> q; *// очередь для хранения узлов для обхода*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов*

        std::**unordered\_set**<**Node** \*> visited; *// множество посещенных узлов*

        std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

        q.**push**(start); *// добавление стартового узла в очередь*

        visited.**insert**(start); *// пометка стартового узла как посещенного*

        parent**[**start**]** = nullptr; *// установка родителя стартового узла как nullptr*

        while (!q.**empty**())

        { *// пока очередь не пуста*

**Node** \*current = q.**front**(); *// получение текущего узла из очереди*

            q.**pop**(); *// удаление текущего узла из очереди*

            if (current == finish)

            { *// если достигнут конечный узел*

                while (current)

                {

                    path.**push\_back**(current); *// добавление текущего узла в путь*

                    current = parent**[**current**]**; *// переход к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// обратный порядок пути (от конечного к стартовому)*

                return path; *// возвращение найденного пути*

            }

            for (auto [neighbor, \_] : current->edges)

            { *// проход по всем соседям текущего узла*

                if (visited.**find**(neighbor) **==** visited.**end**())

                { *// если сосед еще не посещен*

                    visited.**insert**(neighbor); *// пометка соседа как посещенного*

                    parent**[**neighbor**]** = current; *// установка текущего узла как родителя соседа*

                    q.**push**(neighbor); *// добавление соседа в очередь для дальнейшего обхода*

                }

            }

        }

        return path; *// возвращение пустого пути, если конечный узел не найден*

    }

    std::**vector**<**Node** \*> **dfs**(**Node** \*start, **Node** \*finish)

    {

        std::**stack**<**Node** \*> s; *// создаем стек для хранения узлов во время обхода графа*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов (для восстановления пути)*

        std::**unordered\_set**<**Node** \*> visited; *// множество для хранения посещенных узлов*

        std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

        s.**push**(start); *// добавляем стартовый узел в стек для начала обхода*

        visited.**insert**(start); *// помечаем стартовый узел как посещенный*

        parent**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr (нет родителя)*

        while (!s.**empty**())

        { *// пока стек не пустой*

**Node** \*current = s.**top**(); *// получаем текущий узел из вершины стека*

            s.**pop**(); *// удаляем текущий узел из стека*

            if (current == finish)

            { *// если достигнут конечный узел*

                while (current)

                { *// восстанавливаем путь от конечного до стартового узла*

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = parent**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// оборачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, \_] : current->edges)

            { *// проходим по всем соседям текущего узла*

                if (visited.**find**(neighbor) **==** visited.**end**())

                { *// если сосед еще не посещен*

                    visited.**insert**(neighbor); *// помечаем соседа как посещенного*

                    parent**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                    s.**push**(neighbor); *// добавляем соседа в стек для дальнейшего обхода*

                }

            }

        }

        return path; *// возвращаем пустой путь, если конечный узел не найден*

    }

    std::**vector**<**Node** \*> **dijkstra**(**Node** \*start, **Node** \*finish)

    {

        std::**priority\_queue**<std::**pair**<double, **Node** \*>, std::**vector**<std::**pair**<double, **Node** \*>>, std::**greater**<>> pq; *// создаем приоритетную очередь для хранения узлов с их расстояниями*

        std::**unordered\_map**<**Node** \*, double> distances; *// хранилище для расстояний от стартового узла до каждого узла*

        std::**unordered\_map**<**Node** \*, **Node** \*> parent; *// хранилище для отслеживания родительских узлов*

        for (auto &node : nodes)

        {

            distances**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем расстояния до всех узлов как бесконечность*

        }

        distances**[**start**]** = 0; *// расстояние до стартового узла равно 0*

        pq.**push**({0, start}); *// добавляем стартовый узел в очередь с расстоянием 0*

        parent**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr*

        while (!pq.**empty**())

        {

            auto [current\_dist, current] = pq.**top**(); *// получаем узел с наименьшим расстоянием из очереди*

            pq.**pop**(); *// удаляем этот узел из очереди*

            if (current\_dist > distances**[**current**]**)

                continue; *// если текущее расстояние больше уже известного, пропускаем этот узел*

            if (current == finish)

            {

                std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

                while (current)

                {

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = parent**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// оборачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, edge\_dist] : current->edges)

            {

                double new\_dist = current\_dist + edge\_dist; *// вычисляем новое расстояние до соседа*

                if (new\_dist < distances**[**neighbor**]**)

                {

                    distances**[**neighbor**]** = new\_dist; *// обновляем расстояние до соседа*

                    pq.**push**({new\_dist, neighbor}); *// добавляем соседа в очередь с обновленным расстоянием*

                    parent**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                }

            }

        }

        return {}; *// возвращаем пустой вектор, если путь не найден*

    }

*// вычисление расстояния между двумя узлами*

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

    {

        return **sqrt**(**pow**(a->lat - b->lat, 2) + **pow**(a->lon - b->lon, 2)); *// вычисляем эвристическую функцию*

    }

    std::**vector**<**Node** \*> **a\_star**(**Node** \*start, **Node** \*finish)

    {

        std::**priority\_queue**<std::**pair**<double, **Node** \*>, std::**vector**<std::**pair**<double, **Node** \*>>, std::**greater**<>> open\_set; *// приоритетная очередь для хранения открытых узлов с их оценками f(n)*

        std::**unordered\_map**<**Node** \*, double> g\_score; *// хранилище для g(n) - стоимости пути от стартового узла до каждого узла*

        std::**unordered\_map**<**Node** \*, double> f\_score; *// хранилище для f(n) - полной стоимости пути через узел (g(n) + h(n))*

        std::**unordered\_map**<**Node** \*, **Node** \*> came\_from; *// хранилище для отслеживания родительских узлов*

*// инициализируем g(n) и f(n) для всех узлов как бесконечность*

        for (const auto &node : nodes)

        {

            g\_score**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем g(n) как бесконечность*

            f\_score**[**node.second.**get**()**]** = std::**numeric\_limits**<double>::**infinity**(); *// инициализируем f(n) как бесконечность*

        }

        g\_score**[**start**]** = 0; *// стоимость пути от стартового узла до самого себя равна 0*

        f\_score**[**start**]** = **heuristic**(start, finish); *// вычисляем начальное значение f(n) для стартового узла*

        open\_set.**push**({f\_score**[**start**]**, start}); *// добавляем стартовый узел в открытый список с его оценкой f(n)*

        came\_from**[**start**]** = nullptr; *// устанавливаем родителя стартового узла как nullptr*

        while (!open\_set.**empty**())

        {

**Node** \*current = open\_set.**top**().second; *// получаем узел с наименьшей оценкой f(n)*

            open\_set.**pop**(); *// удаляем этот узел из открытого списка*

            if (current == finish)

            {

                std::**vector**<**Node** \*> path; *// вектор для хранения найденного пути*

                while (current)

                {

                    path.**push\_back**(current); *// добавляем текущий узел в путь*

                    current = came\_from**[**current**]**; *// переходим к родительскому узлу*

                }

**reverse**(path.**begin**(), path.**end**()); *// разворачиваем путь, чтобы он был от стартового до конечного узла*

                return path; *// возвращаем найденный путь*

            }

            for (auto [neighbor, weight] : current->edges) *// проходим по всем соседям текущего узла*

            {

                double tentative\_g\_score = g\_score**[**current**]** + weight; *// вычисляем предполагаемую стоимость пути до соседа через текущий узел*

                if (tentative\_g\_score < g\_score**[**neighbor**]**)

                {

                    came\_from**[**neighbor**]** = current; *// устанавливаем текущий узел как родителя соседа*

                    g\_score**[**neighbor**]** = tentative\_g\_score; *// обновляем g(n) для соседа*

                    f\_score**[**neighbor**]** = g\_score**[**neighbor**]** + **heuristic**(neighbor, finish); *// обновляем f(n) для соседа*

                    open\_set.**push**({f\_score**[**neighbor**]**, neighbor}); *// добавляем соседа в открытый список с обновленным значением f(n)*

                }

            }

        }

        return {}; *// возвращаем пустой вектор, если путь не найден*

    }

*// вывод узлов и длины пути*

    void **print\_path**(std::**vector**<**Node** \*> &path)

    {

        if (path.**empty**())

        {

            std::cout **<<** "Путь не найден." **<<** std::**endl**;

            return;

        }

*/\* for (Node \*n : path)*

*{*

*std::cout << n->lon << " " << n->lat << std::endl; // выводим координаты узла*

*} \*/*

        std::cout **<<** path.**size**() **<<** " узлов" **<<** std::**endl**; *// выводим длину пути*

    }

*// подсчёт времени bfs*

    void **time\_bfs**(**Node** \*start\_node, **Node** \*finish\_node)

    {

        auto start\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время начала*

        std::**vector**<**Node** \*> path = **bfs**(start\_node, finish\_node); *// вызываем bfs для поиска пути*

        auto end\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время окончания*

        std::cout **<<** "BFS: ";

**print\_path**(path); *// выводим найденный путь*

        std::cout **<<** "время BFS: " **<<** std::chrono::**duration**<double>(end\_time **-** start\_time).**count**() **<<** " сек" **<<** std::**endl**; *// выводим время выполнения*

    }

    void **time\_dfs**(**Node** \*start\_node, **Node** \*finish\_node)

    {

        auto start\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время начала*

        std::**vector**<**Node** \*> path = **dfs**(start\_node, finish\_node); *// вызываем dfs для поиска пути*

        auto end\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время окончания*

        std::cout **<<** "DFS: ";

**print\_path**(path); *// выводим найденный путь*

        std::cout **<<** "время DFS: " **<<** std::chrono::**duration**<double>(end\_time **-** start\_time).**count**() **<<** " сек" **<<** std::**endl**; *// выводим время выполнения*

    }

    void **time\_dijkstra**(**Node** \*start\_node, **Node** \*finish\_node)

    {

        auto start\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время начала*

        std::**vector**<**Node** \*> path = **dijkstra**(start\_node, finish\_node); *// вызываем алгоритм Дейкстры для поиска пути*

        auto end\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время окончания*

        std::cout **<<** "Dijkstra: ";

**print\_path**(path); *// выводим найденный путь*

        std::cout **<<** "время Dijkstra: " **<<** std::chrono::**duration**<double>(end\_time **-** start\_time).**count**() **<<** " сек" **<<** std::**endl**; *// выводим время выполнения*

    }

    void **time\_a\_star**(**Node** \*start\_node, **Node** \*finish\_node)

    {

        auto start\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время начала*

        std::**vector**<**Node** \*> path = **a\_star**(start\_node, finish\_node); *// вызываем A\* для поиска пути*

        auto end\_time = std::chrono::**high\_resolution\_clock**::**now**(); *// фиксируем время окончания*

        std::cout **<<** "A\*: ";

**print\_path**(path); *// выводим найденный путь*

        std::cout **<<** "время A\*: " **<<** std::chrono::**duration**<double>(end\_time **-** start\_time).**count**() **<<** " сек" **<<** std::**endl**; *// выводим время выполнения*

    }

};

void **run\_spb\_graph**()

{

**Graph** graph; *// создаем объект графа*

    graph.**load\_graph\_from\_file**("spb\_graph.txt"); *// загружаем граф из файла*

    double start\_lon = 30.48455, start\_lat = 59.95878; *// дом*

    double finish\_lon = 30.295483, finish\_lat = 59.944077; *// биржа*

**Node** \*start\_node = graph.**find\_closest\_node**(start\_lat, start\_lon); *// находим ближайший узел к начальной точке*

**Node** \*finish\_node = graph.**find\_closest\_node**(finish\_lat, finish\_lon); *// находим ближайший узел к конечной точке*

    graph.**time\_bfs**(start\_node, finish\_node); *// подсчитываем время BFS*

    std::cout **<<** std::**endl**;

    graph.**time\_dfs**(start\_node, finish\_node); *// подсчитываем время DFS*

    std::cout **<<** std::**endl**;

    graph.**time\_dijkstra**(start\_node, finish\_node); *// подсчитываем время Dijkstra*

    std::cout **<<** std::**endl**;

    graph.**time\_a\_star**(start\_node, finish\_node); *// подсчитываем время A\**

    std::cout **<<** std::**endl**;

};

int **main**()

{

**run\_spb\_graph**(); *// запускаем функцию для работы с графом*

    return 0; *// завершаем программу*

}